Transaction-Level Models of Systems-on-a-Chip
Can they be Fast, Correct and Faithful?

Matthieu Moy

Laboratoire d'Informatique du Parallelisme
Lyon, France

February 2018

About me

2005	Ph.D: formal verification of SoC models (ST/Verimag)
2006	Post-doc: security of storage (Bangalore, Inde)
2006	Assistant professor, Verimag / Ensimag Work on SoC models & abstract interpretation
2014	HDR: High-Level models for Embedded Systems
2017	New CASH team leader, LIP / UCBL

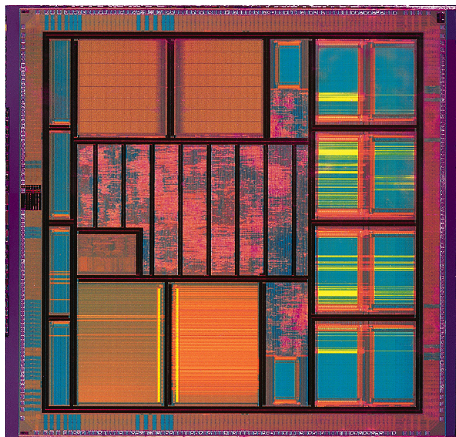
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion

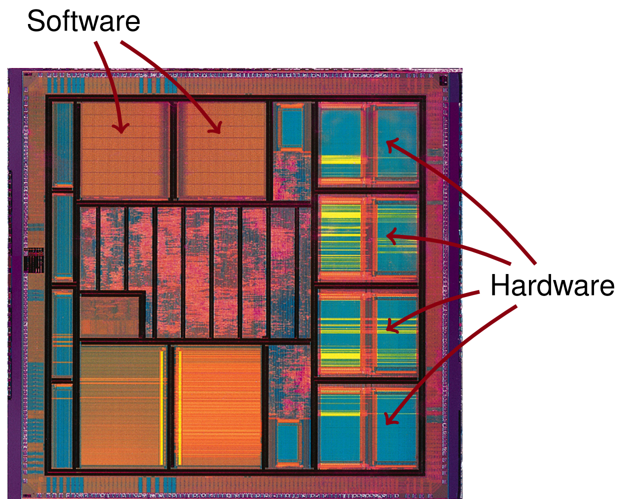
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion

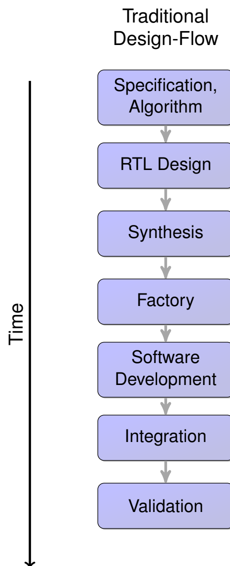
Modern Systems-on-a-Chip



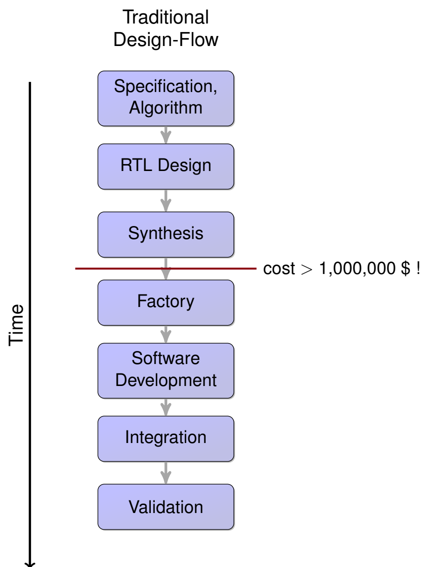
Modern Systems-on-a-Chip



Hardware/Software Design Flow

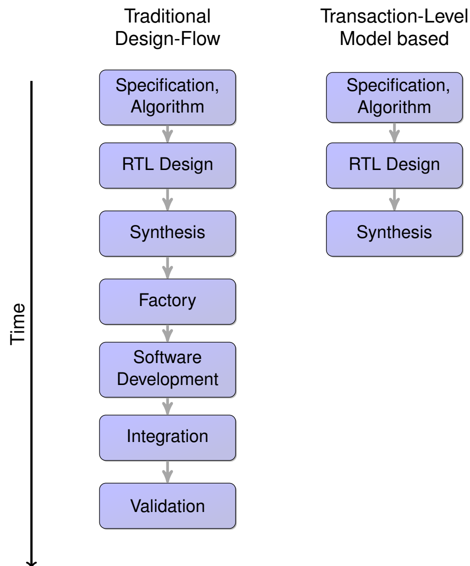


Hardware/Software Design Flow

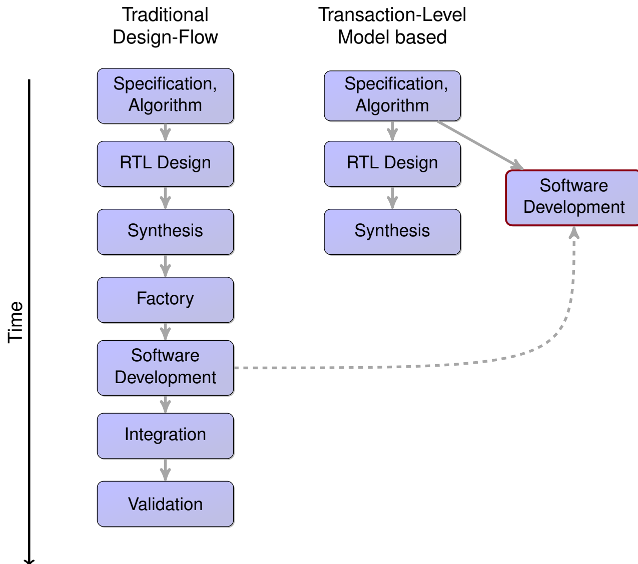


cost > 1,000,000 $!

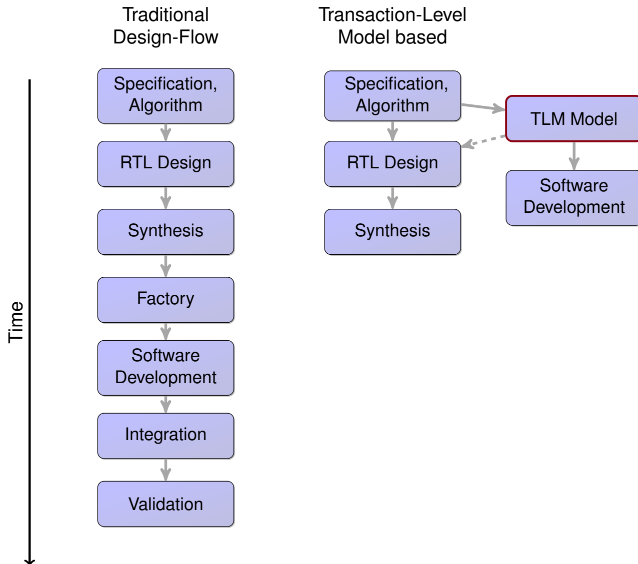
Hardware/Software Design Flow



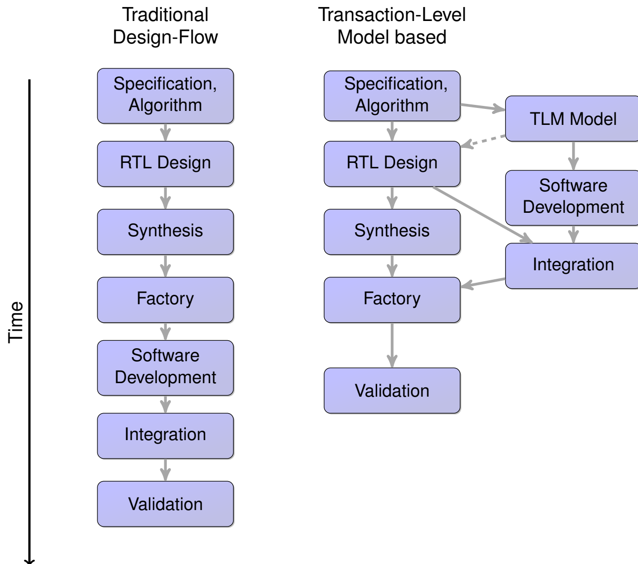
Hardware/Software Design Flow



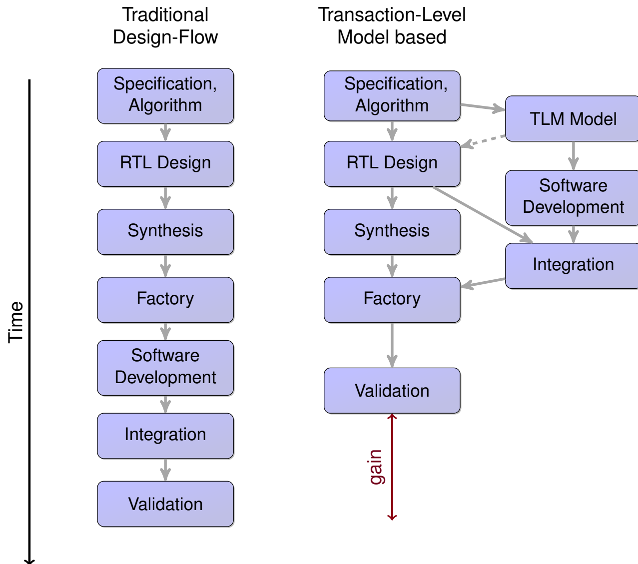
Hardware/Software Design Flow



Hardware/Software Design Flow



Hardware/Software Design Flow



The Transaction Level Model: Principles and Objectives

A high level of abstraction,
that appears early in the design-flow

The Transaction Level Model: Principles and Objectives

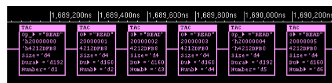
A high level of abstraction,
that appears early in the design-flow

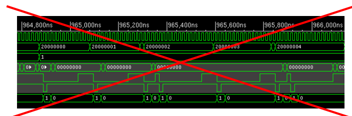
- A virtual prototype of the system, to enable
  - Early software development
  - Integration of components
  - Architecture exploration
  - Reference model for validation
- Abstract implementation details from RTL
  - Fast simulation ($\simeq$ 1000x faster than RTL)
  - Lightweight modeling effort ($\simeq$ 10x less than RTL)

Content of a TLM Model

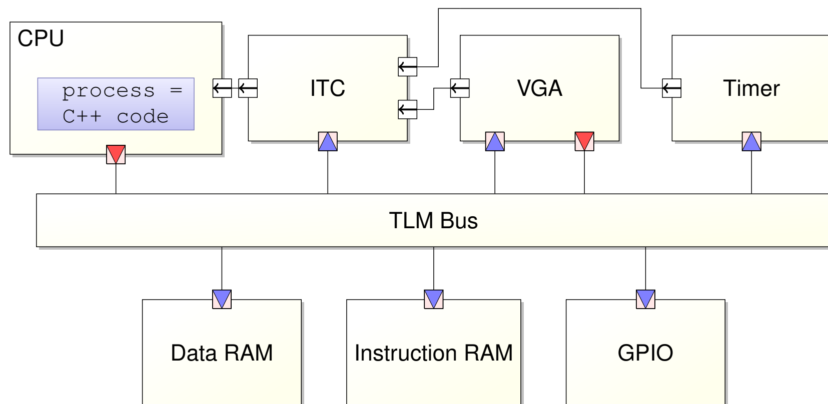
A first definition

- Model what is needed for Software Execution:
  - Processors
  - Address-map
  - Concurrency
- ... and only that.
  - No micro-architecture
  - No bus protocol
  - No pipeline
  - No physical clock
  - ...

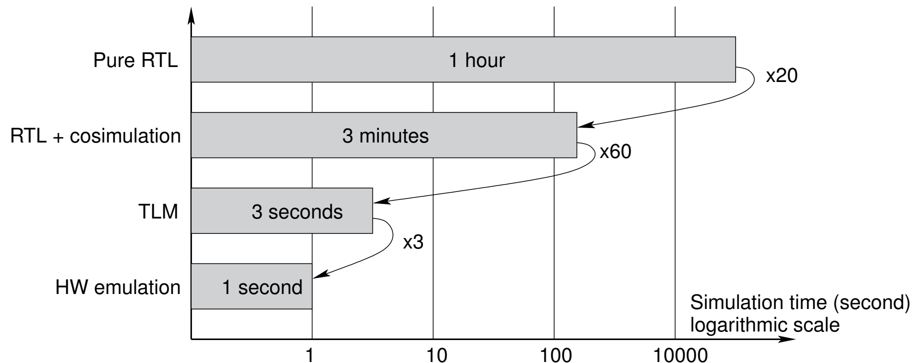




An example TLM Model



Performance of TLM



Uses of Functional Models

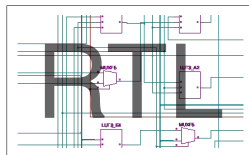
Reference for Hardware Validation

Virtual Prototype for Software Development

Uses of Functional Models

Reference for Hardware Validation

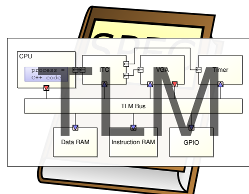
$\stackrel{?}{=}$



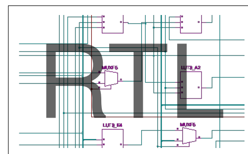
Virtual Prototype for Software Development

Uses of Functional Models

Reference for Hardware Validation



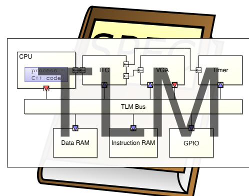
$\stackrel{?}{=}$



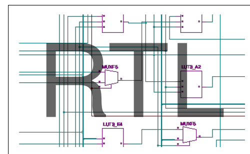
Virtual Prototype for Software Development

Uses of Functional Models

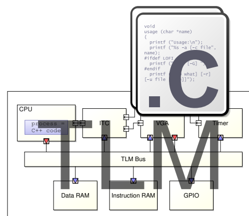
Reference for Hardware Validation



$\stackrel{?}{=}$

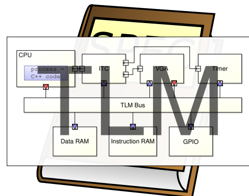


Virtual Prototype for Software Development

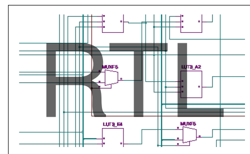


Uses of Functional Models

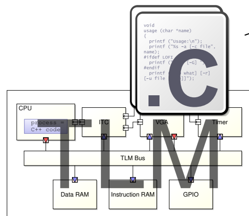
Reference for Hardware Validation



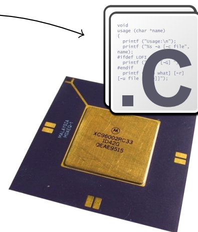
$\stackrel{?}{=}$



Virtual Prototype for Software Development

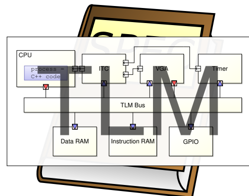


Unmodified Software

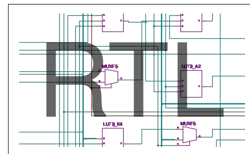


Uses of Functional Models

Reference for Hardware Validation

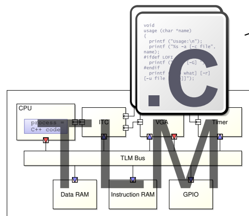


$\stackrel{?}{=}$

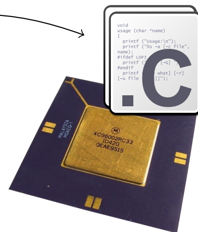


Unmodified Software

Virtual Prototype for Software Development



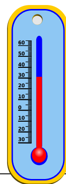
$\stackrel{?}{\supseteq}$



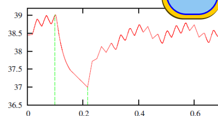
Content of a TLM Model

A richer definition

- Timing information
  - May be needed for Software Execution
  - Useful for Profiling Software



- Power and Temperature
  - Validate design choices
  - Validate power-management policy

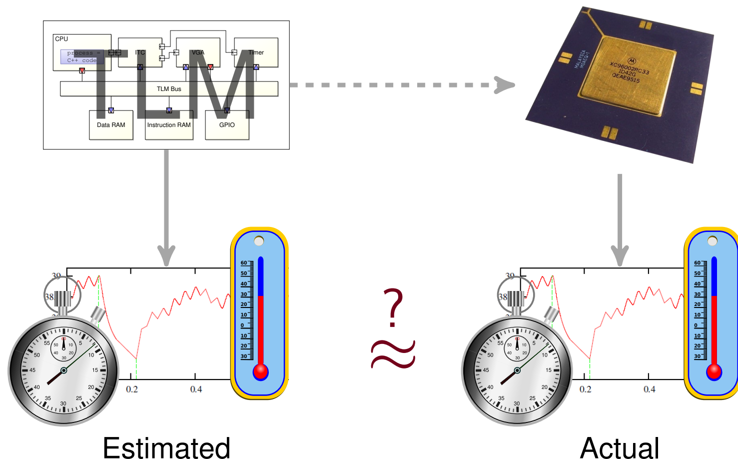


Use of Extra-Functional Models

Timing, Power consumption, Temperature Estimation

Use of Extra-Functional Models

Timing, Power consumption, Temperature Estimation



Use of Extra-Functional Models

Timing, Power consumption, Temperature Estimation

Unmodified
Power/Temperature-Aware
Software

CPU
TLM Bus
Data RAM
Instruction RAM
GPIO

?
≈

Estimated

Actual

Summary: Expected Properties of TLM Programs

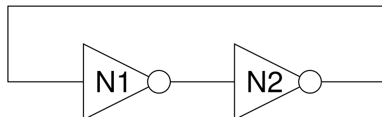
SystemC/TLM Programs should

- Simulate fast,
- Satisfy correctness criterions,
- Reflect faithfully functional and extra-functional properties of the actual system.

Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion

SystemC: Simple Example




```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;

    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    // Instantiate modules ...
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // ... and bind them together
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS);
    return 0;
}
```

Compiling SystemC

```
$ g++ example.cpp -lsystemc
$ ./a.out
```

... end of section?

Compiling SystemC

```
$ g++ example.cpp -lsystemc
$ ./a.out
```

But ...

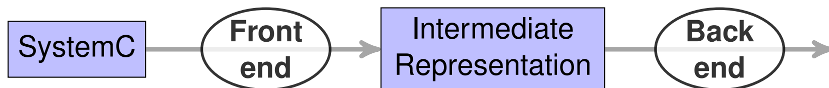
- C++ compilers cannot do SystemC-aware optimizations
- C++ analyzers do not know SystemC semantics

This section

2. Compilation of SystemC/TLM
   - Front-end
   - Optimization and Fast Simulation

SystemC Front-End

- In this talk: Front-end = “Compiler front-end” (AKA “Parser”)



Intermediate Representation = Architecture + Behavior

SystemC Front-Ends

- When you *don't* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...

SystemC Front-Ends

- When you *don't* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...

$\Rightarrow$ No reference front-end available on
`http://www.accellera.org/`

SystemC Front-Ends

- When you *don't* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...

$\Rightarrow$ No reference front-end available on `http://www.accellera.org/`

- When you *do* need a front-end:
  - Symbolic formal verification, High-level synthesis
  - Visualization
  - Introspection
  - SystemC-specific Compiler Optimizations
  - Advanced debugging features

Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)
2. Architecture built at runtime, with C++ code

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)
   $\leadsto$ Write a C++ front-end or reuse one (g++, clang, EDG, ...)
2. Architecture built at runtime, with C++ code
   $\leadsto$ Analyze elaboration phase or execute it

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)
   ⇝ Write a C++ front-end or reuse one (g++, clang, EDG, ...)
2. Architecture built at runtime, with C++ code
   ⇝ Analyze elaboration phase or execute it

```
SC_MODULE(no _gate) {
    sc_in<bo l> in;
    sc_out< ol> out;
    void co ute (void) {
        //  avior
                    in.read();
                          );
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(in  argc, char **argv) {
    // Elaborat   phase (Architecture)
    not_gate n1("
    not_g
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Dealing with the architecture

When it becomes tricky. . .

```
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j]
                = new Node(...);
            ...
        }
    }

    sc_start(100, SC_NS);
    return 0;
}
```

Dealing with the architecture

When it becomes tricky. . .

- Static approach: cannot deal with such code
- Dynamic approach: can extract the architecture for individual instances of the system

```
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j]
                = new Node(...);
            ...
        }
    }

    sc_start(100, SC_NS);
    return 0;
}
```

Dealing with the architecture

When it becomes *very* tricky. . .

```
void compute(void) {
    for (int i = 0; i < n; i++) {
        ports[i].write(true);
    }
    ...
}
```

Dealing with the architecture

When it becomes *very* tricky...

- One can unroll the loop to let `i` become constant,
- Undecidable in the general case.

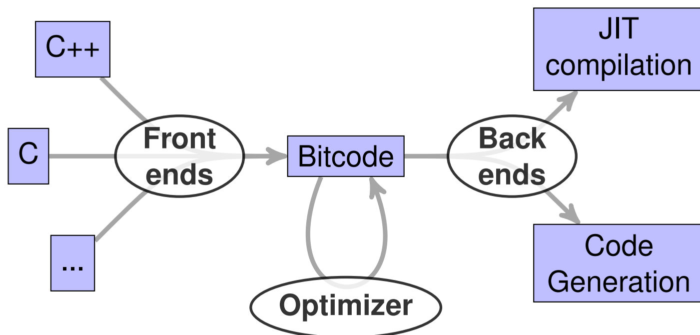
```
void compute(void) {
    for (int i = 0; i < n; i++) {
        ports[i].write(true);
    }
    ...
}
```

The beginning: Pinapa

AKA “my Ph.D’s front-end”

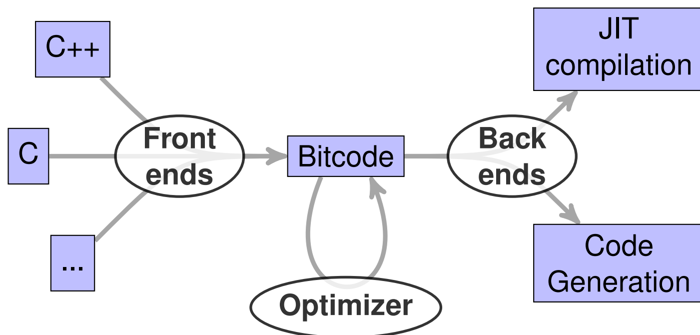
- Pinapa’s principle:
  - Use GCC’s C++ front-end
  - Compile, dynamically load and execute the elaboration (`sc_main`)
- Pinapa’s drawbacks:
  - Uses GCC’s internals (hard to port to newer versions)
  - Hard to install and use, no separate compilation
  - Ad-hoc match of SystemC constructs in AST
  - AST *Vs* SSA form in modern compilers

LLVM: Low Level Virtual Machine

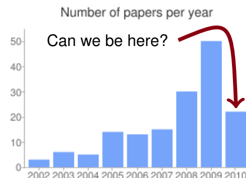


- Clean API
- Clean SSA intermediate representation
- Many tools available

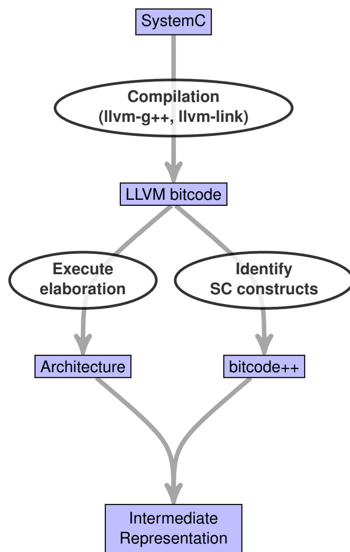
LLVM: Low Level Virtual Machine



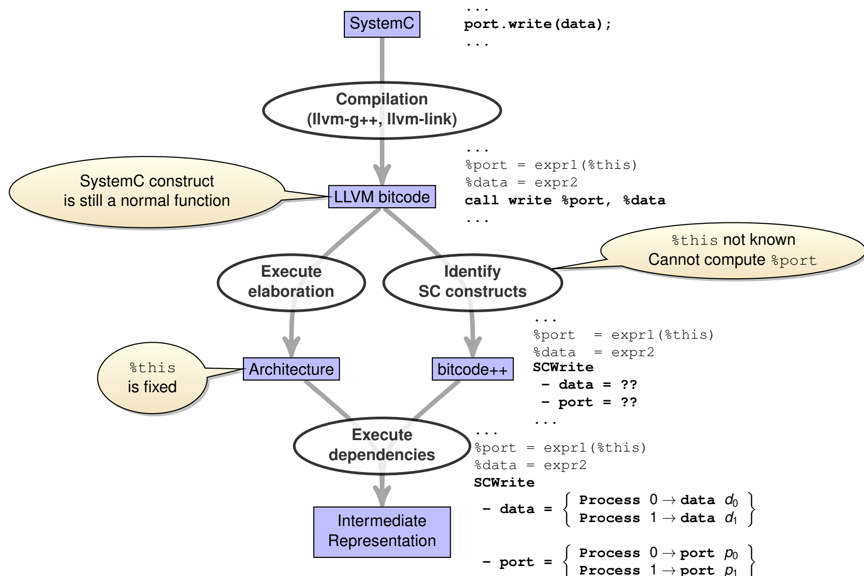
- Clean API
- Clean SSA intermediate representation
- Many tools available



PinaVM: Enriching the bitcode



PinaVM: Enriching the bitcode



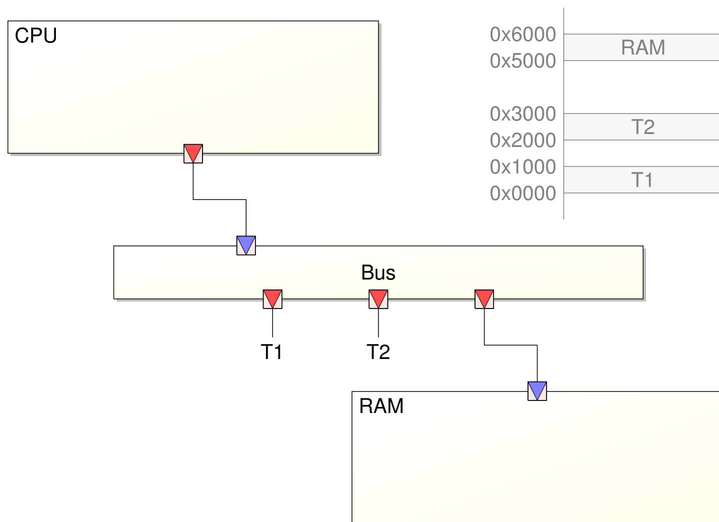
Summary

- PinaVM relies on executability (JIT Compiler) for execution of:
  - elaboration phase ($\approx$ like Pinapa)
  - sliced pieces of code
- Open Source: `http://forge.imag.fr/projects/pinavm/`
- Still a prototype, but very few fundamental limitations
- $\approx$ 3000 lines of C++ code on top of LLVM
- Experimental back-ends for
  - Execution (Tweto)
  - Model-checking (using SPIN)

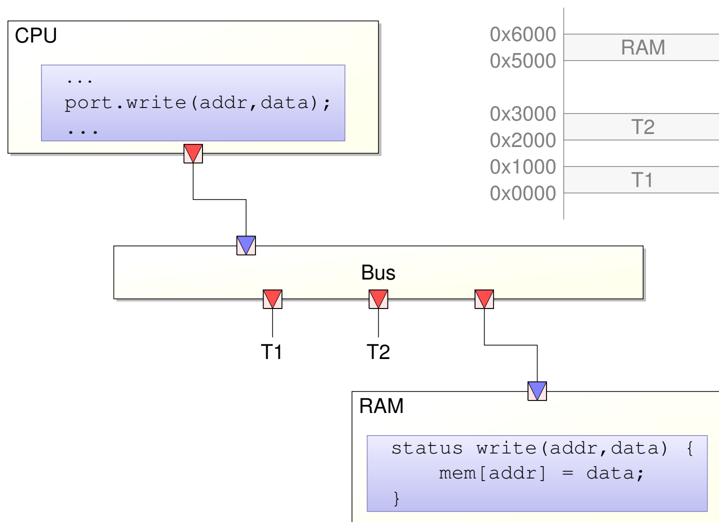
This section

2. Compilation of SystemC/TLM
   - Front-end
   - Optimization and Fast Simulation

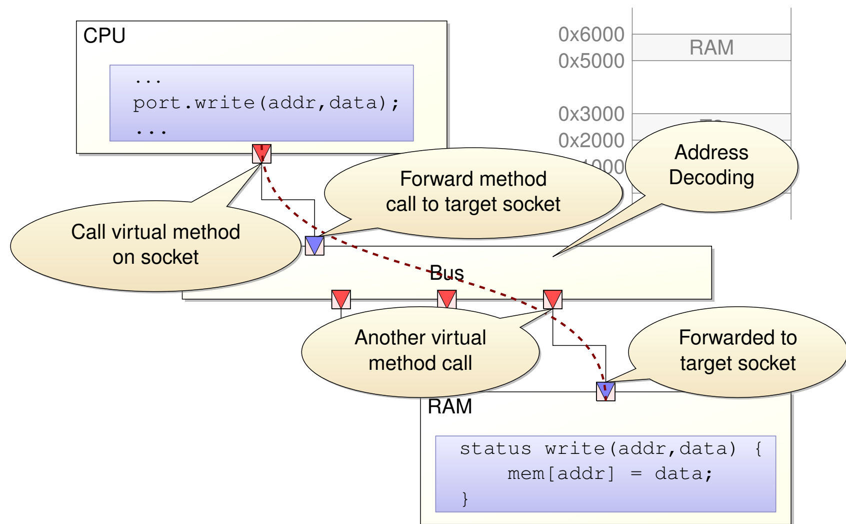
Typical Transaction Journey



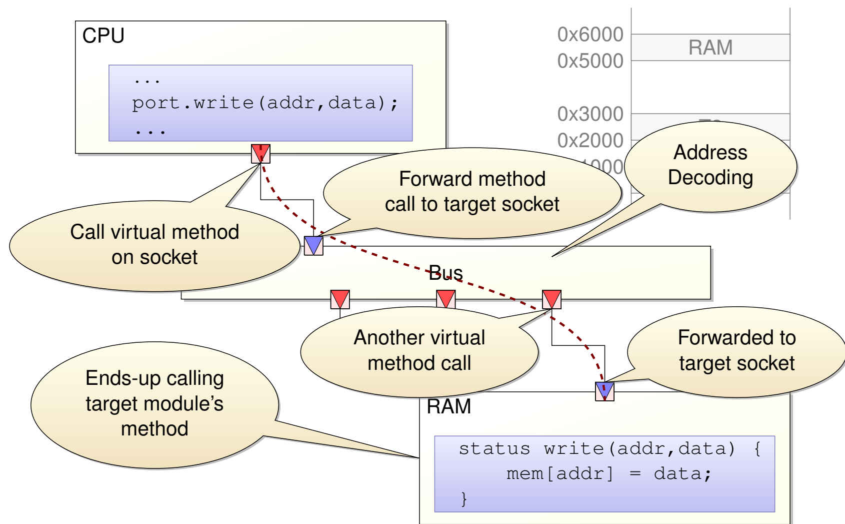
Typical Transaction Journey



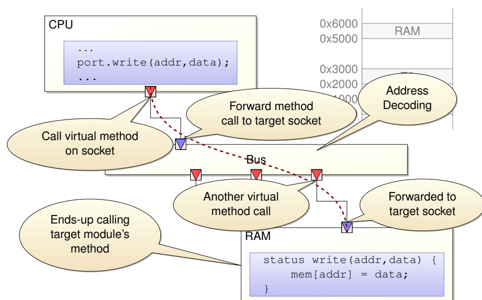
Typical Transaction Journey



Typical Transaction Journey

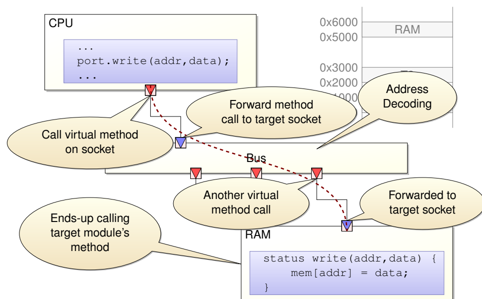


Typical Transaction Journey



- Many costly operations for a simple functionality
- Work-around: backdoor access (DMI = Direct Memory Interface)
  - CPU get a pointer to RAM's internal data
  - Manual, dangerous optimization

Typical Transaction Journey



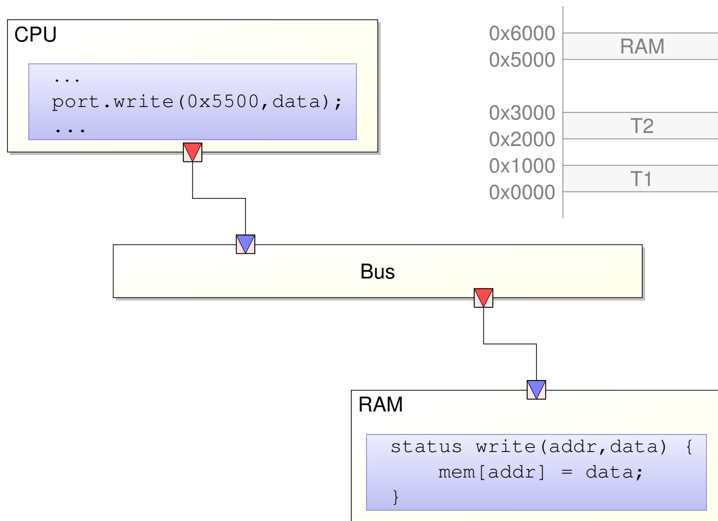
- Many costly operations for a simple functionality
- Work-around: backdoor access (DMI = Direct Memory Interface)
  - CPU get a pointer to RAM's internal data
  - Manual, dangerous optimization

Can a compiler be as good as DMI,
automatically and safely?

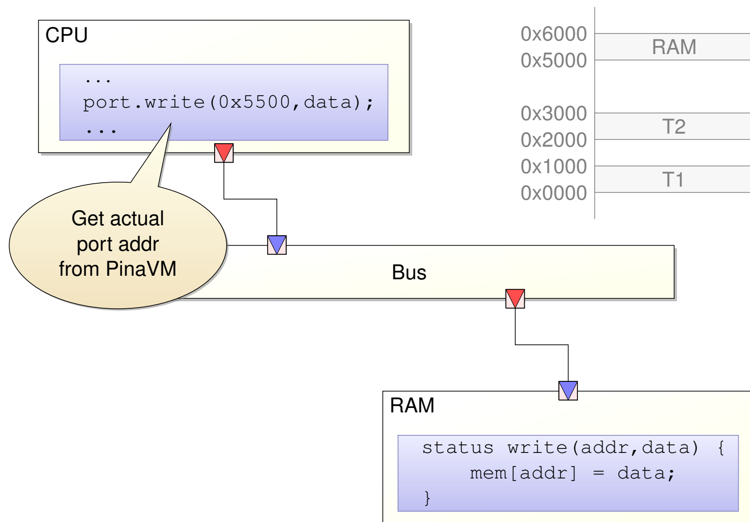
Basic Ideas

- Do statically what can be done statically ...
- ... considering “statically” = “after elaboration”
- Examples:
  - Virtual function resolution
  - Inlining through SystemC ports
  - Static address resolution

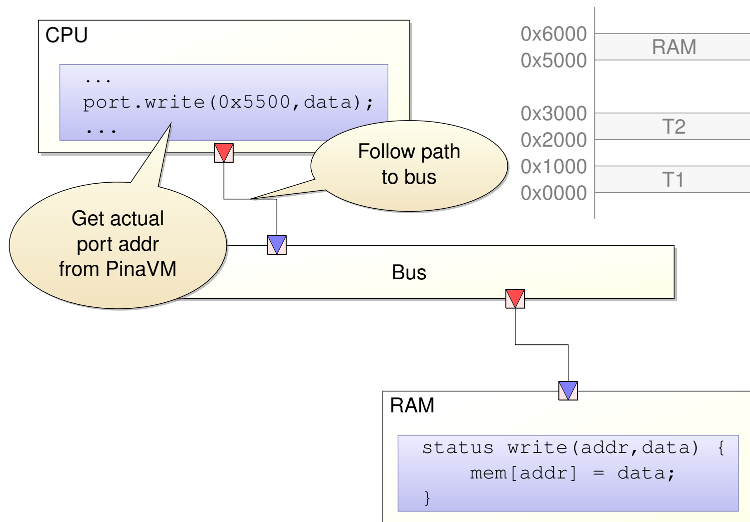
Dealing with addresses *Statically*



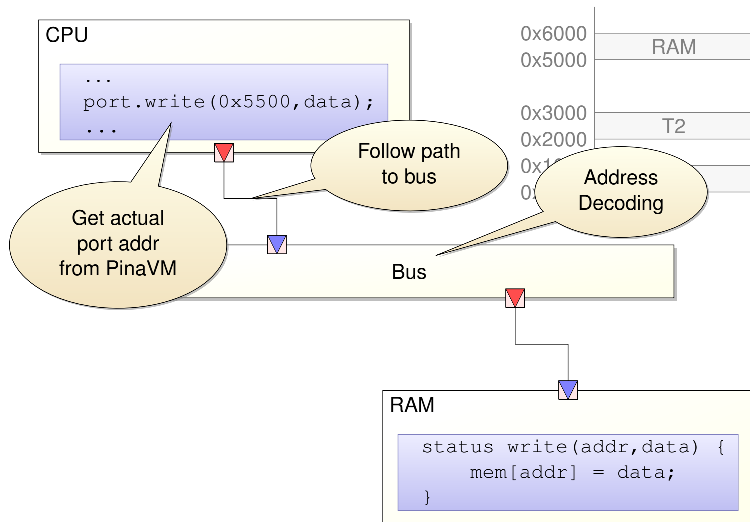
Dealing with addresses *Statically*



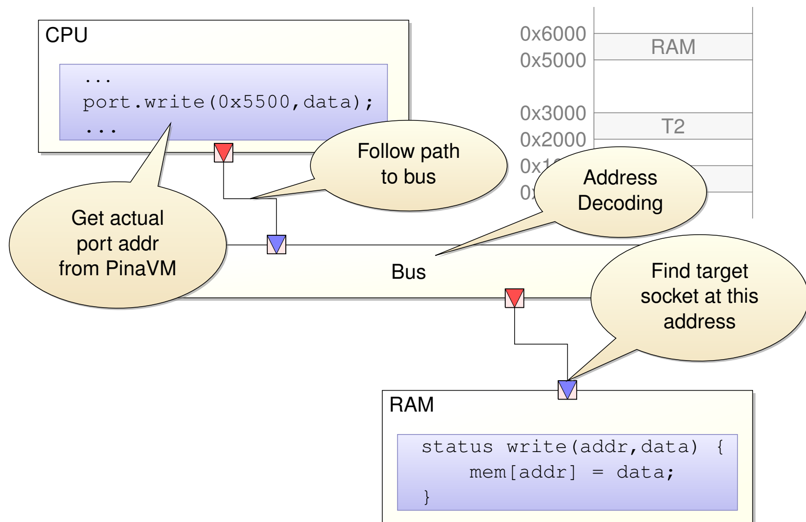
Dealing with addresses *Statically*



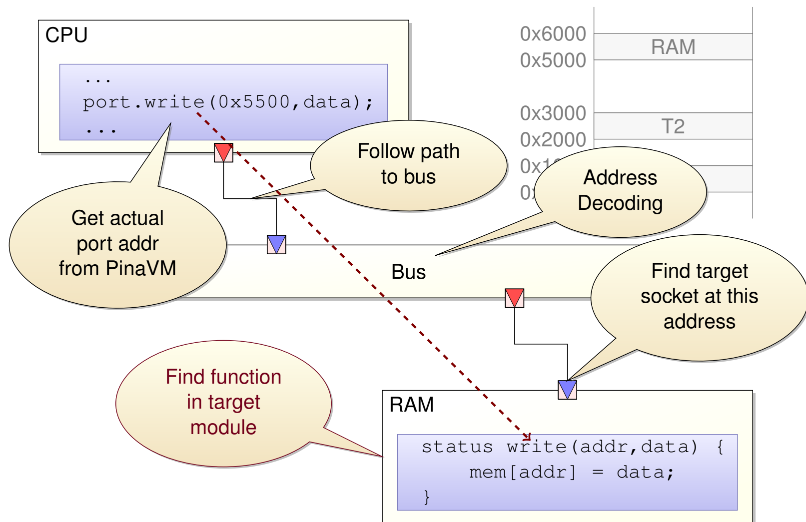
Dealing with addresses *Statically*



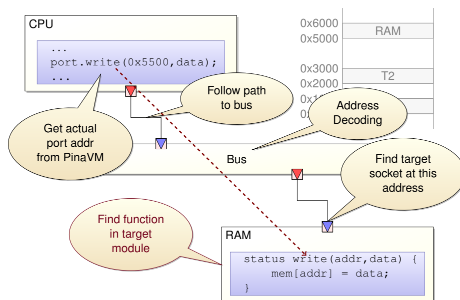
Dealing with addresses *Statically*



Dealing with addresses *Statically*



Dealing with addresses *Statically*

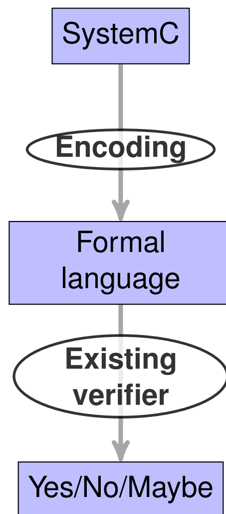


- Possible optimizations:
  - Replace call to `port.write()` with `RAM.write()`
  - Possibly inline it

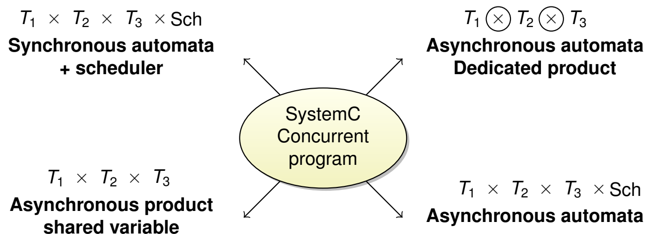
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion

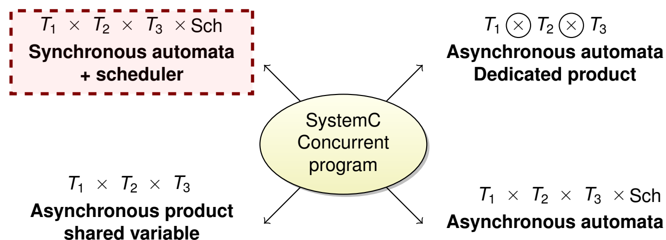
Encoding Approaches



Encoding Approaches



Encoding Approaches

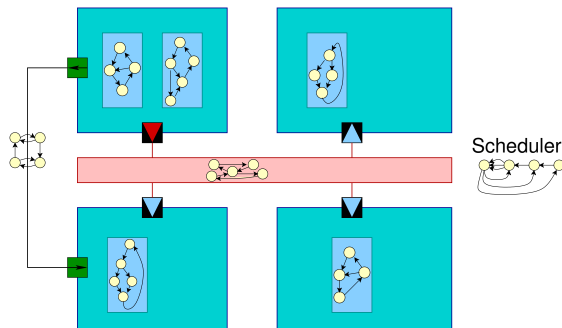


Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton

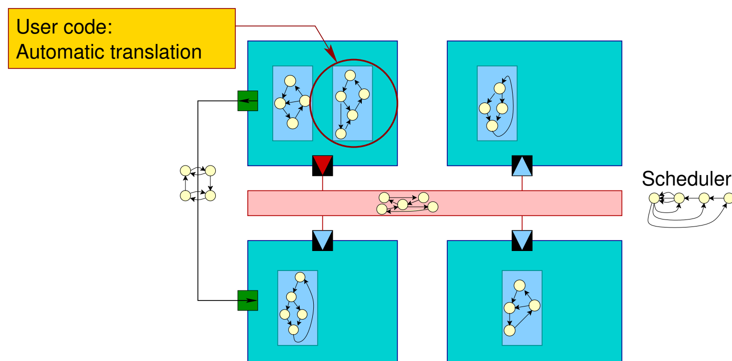
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton



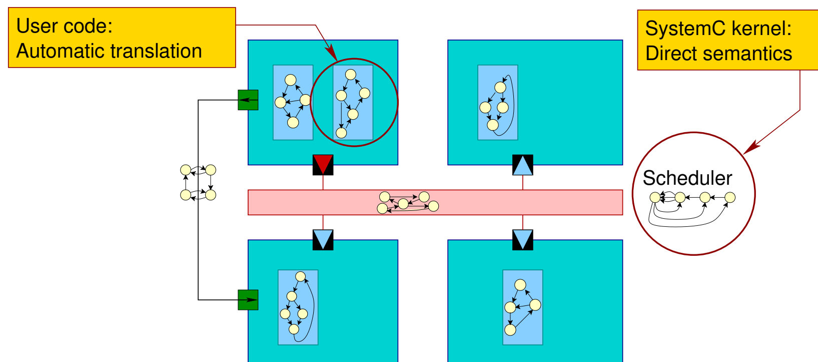
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton



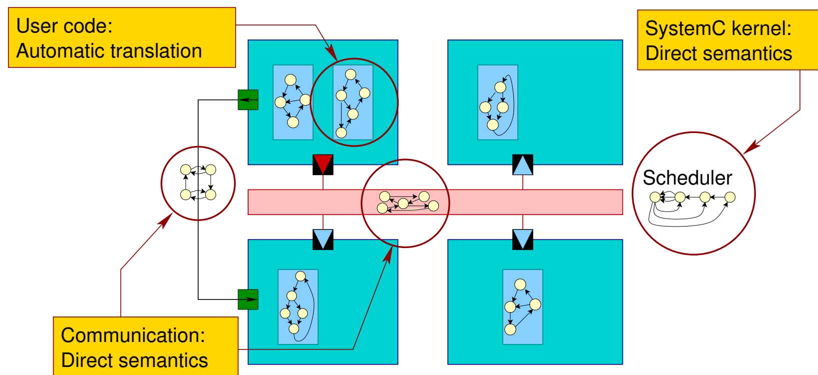
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton



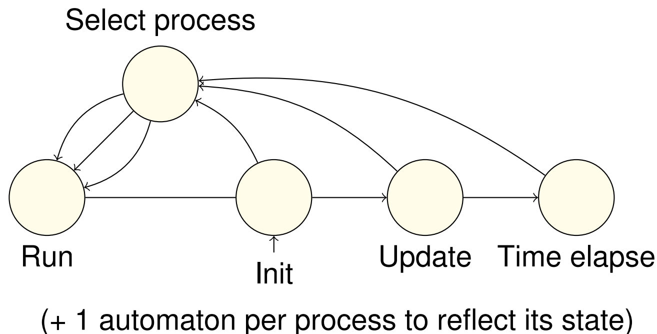
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton



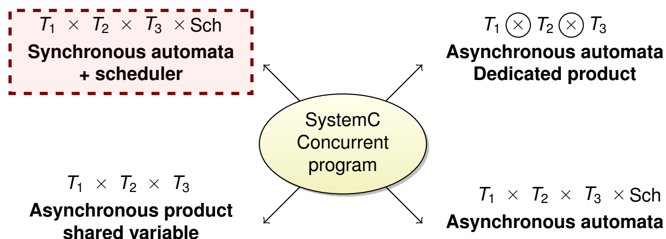
The SystemC scheduler

- Non-preemptive scheduler
- Non-deterministic processes election

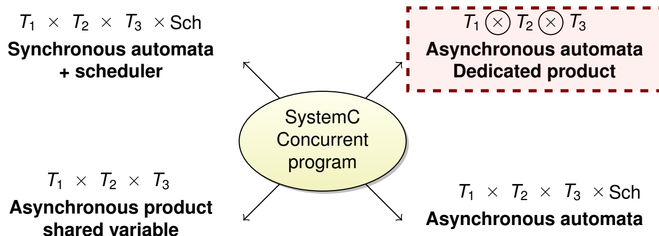


(+ 1 automaton per process to reflect its state)

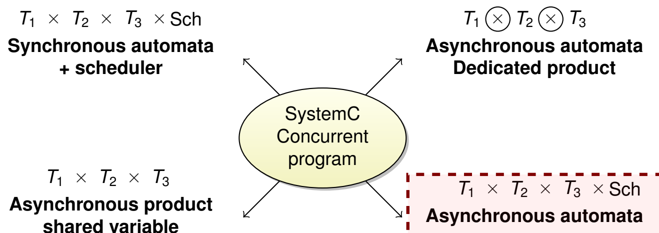
Encoding Approaches



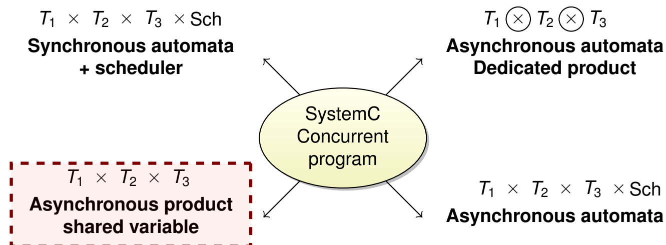
Encoding Approaches



Encoding Approaches



Encoding Approaches



SystemC to Spin: encoding events

- notify/wait for event E^k:

p::wait(E^k):	**p::notify(E^k):**
$W_p := k$	$\forall i \in P \vert W_i == K$
blocked($W_p == 0$)	$W_i := 0$

- W_p : integer associated to process p.
  $W_p = k \Leftrightarrow$ "process p is waiting for event E^k".

SystemC to Spin: encoding time and events

- discrete time
- a deadline variable T_p is attached to each process p
  T_p = next execution time for process p

p::wait(d):
$T_p := T_p + d$
blocked($T_p == \min_{i \in P} (T_i)$)

"Set my next execution time to now $+ d$ and wait until the current execution time reaches it"

SystemC to Spin: encoding time and events

- discrete time
- a deadline variable T_p is attached to each process p
  T_p = next execution time for process p

p::wait(d):

$$T_p := T_p + d$$

$$\text{blocked}(T_p == \min_{\substack{i \in P \\ \mathbf{W_i == 0}}} (T_i))$$

"Set my next execution time to now + d and wait until the current execution time reaches it"

p::wait(E^k):

$$W_p := \text{K}$$

$$\text{blocked}(W_p == 0)$$

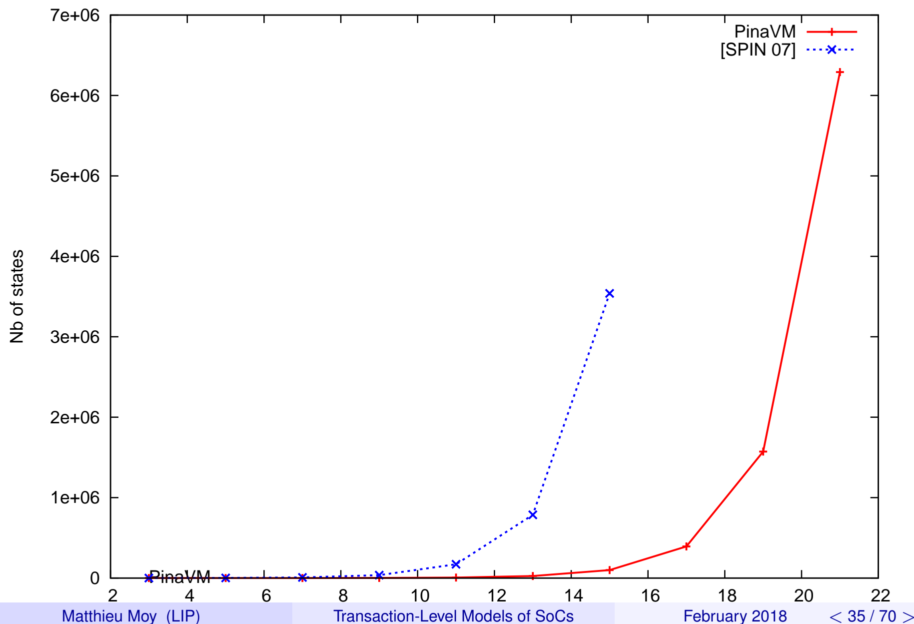
p::notify(E^k):

$$\forall i \in P | W_i == K$$

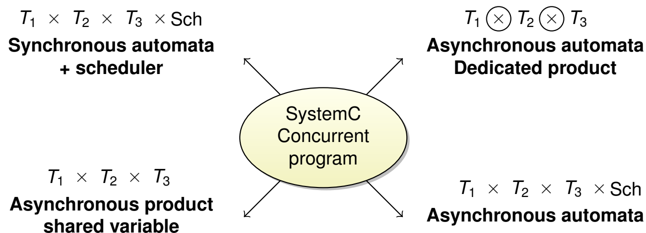
$$W_i := 0$$

$$T_i := T_p$$

SystemC to Spin: results



Encoding Approaches



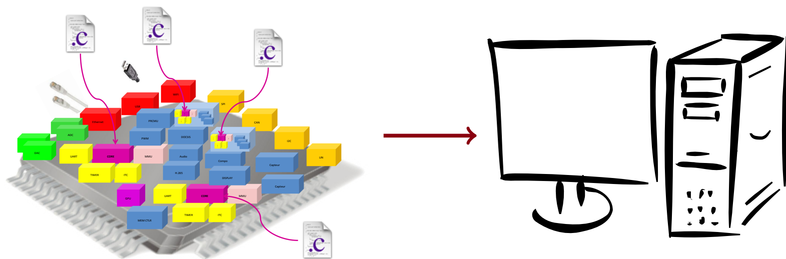
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion

This section

4 Extra-Functional Properties in TLM
- Time and Parallelism
- Power and Temperature Estimation

Parallelization of Simulations



Parallelization of Simulations

System-level Simulation Vs HPC

Problems and solutions for parallel execution of SystemC/TLM

(1) Execution order imposed by SystemC semantics

(2) Concurrent access to shared resources
(e.g., `x++` on a global variable)

Problems and solutions for parallel execution of SystemC/TLM

(1) Execution order imposed by SystemC semantics

(2) Concurrent access to shared resources
(e.g., `x++` on a global variable)

$\leadsto$ No 100% automatic and efficient solution for TLM

Problems and solutions for parallel execution of SystemC/TLM

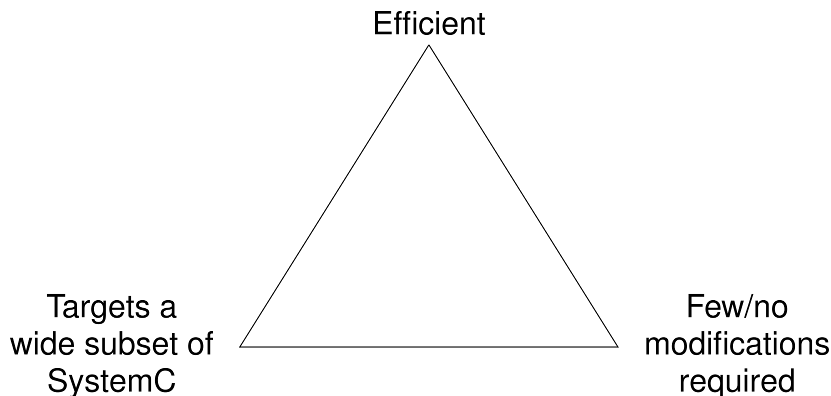
(1) Execution order imposed by SystemC semantics

(2) Concurrent access to shared resources
(e.g., `x++` on a global variable)

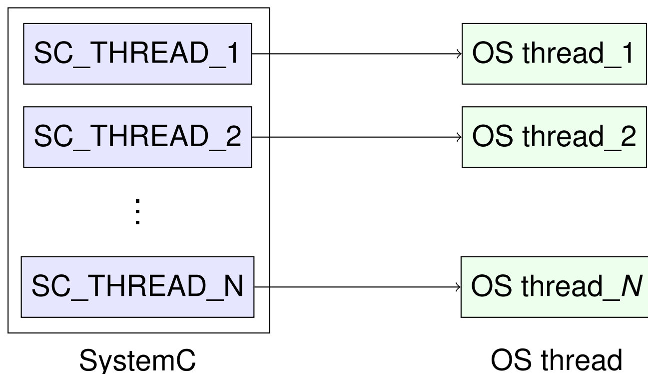
$\rightsquigarrow$ No 100% automatic and efficient solution for TLM

Our proposal = additional constructs:
Desynchronization (1) / Synchronization (2)

Approaches to parallelization

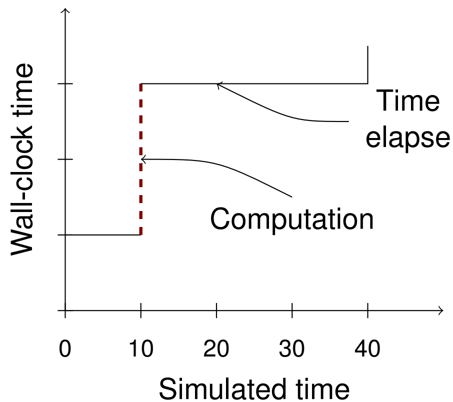


SC-DURING: The Idea

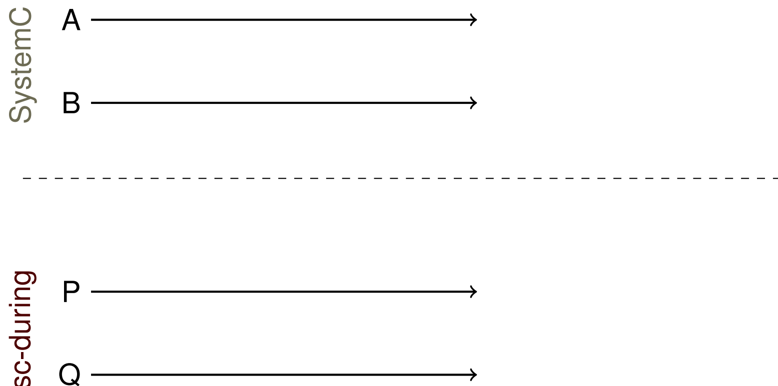


- Unmodified SystemC
- Some computation delegated to other threads
- Weak synchronization between SystemC and threads thanks to tasks with duration

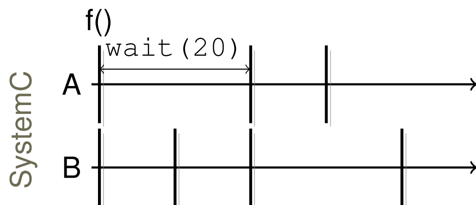
Simulated Time Vs Wall-Clock Time



Simulated Time in SystemC and SC-DURING

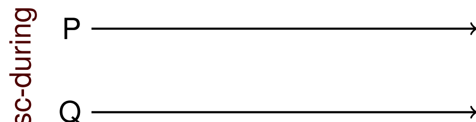


Simulated Time in SystemC and SC-DURING

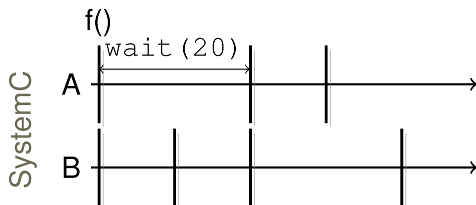


Process A:

```
//Computation
f();
//Time taken by f
wait(20);
```

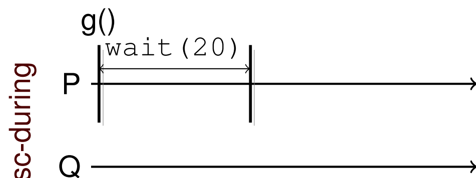


Simulated Time in SystemC and SC-DURING



Process A:

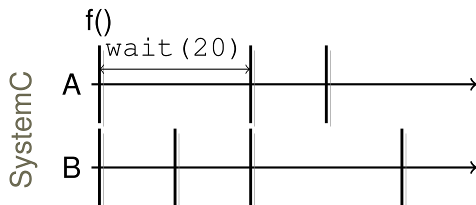
```
//Computation
f();
//Time taken by f
wait(20);
```



Process P:

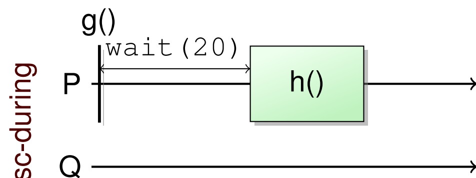
```
g();
wait(20);
```

Simulated Time in SystemC and SC-DURING



Process A:

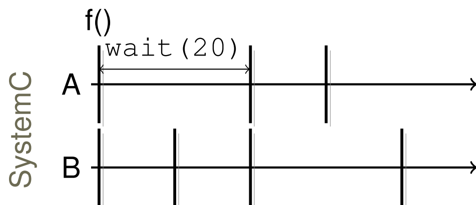
```
//Computation
f();
//Time taken by f
wait(20);
```



Process P:

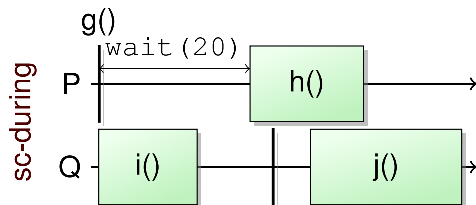
```
g();
wait(20);
during(15, h);
```

Simulated Time in SystemC and SC-DURING



Process A:

```
//Computation
f();
//Time taken by f
wait(20);
```



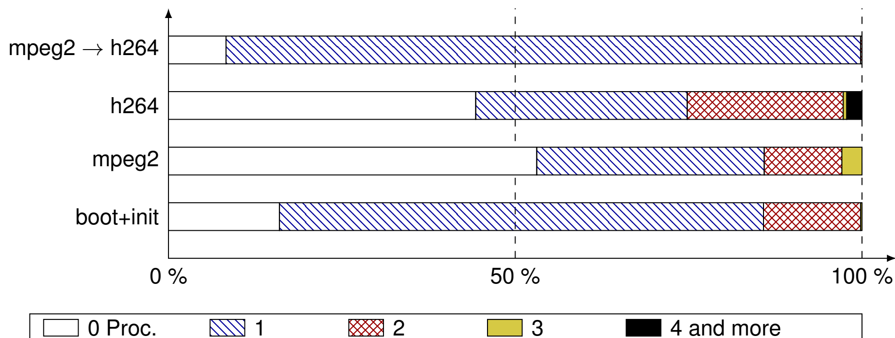
Process P:

```
g();
wait(20);
during(15, h);
```

Impact on Parallelism

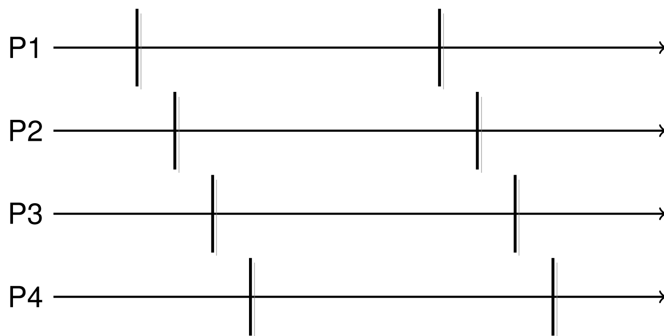
Concurrency in an industrial platform

Number of SystemC threads active within a cycle (ST set-top-box case study) :

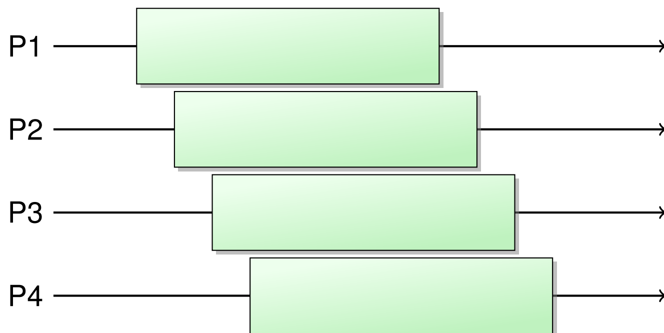


Impact on Parallelism

Impact on Parallelism

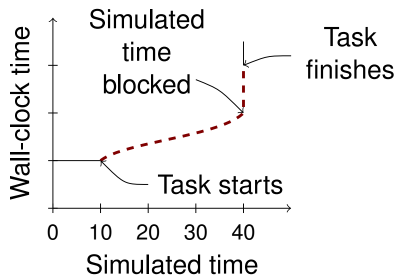


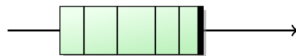
Impact on Parallelism

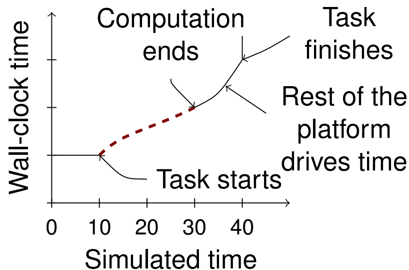


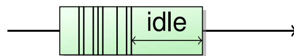
Overlap between tasks $\leadsto$ parallel execution in `sc-during`

Execution of `during(T)`



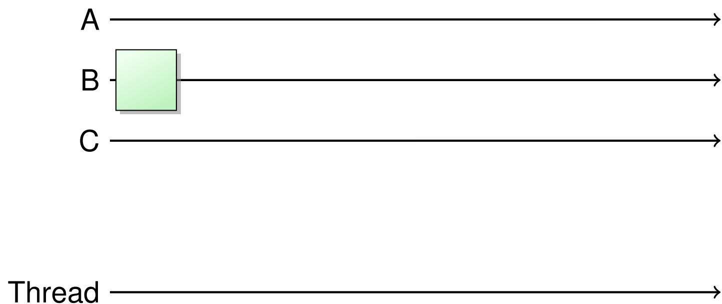






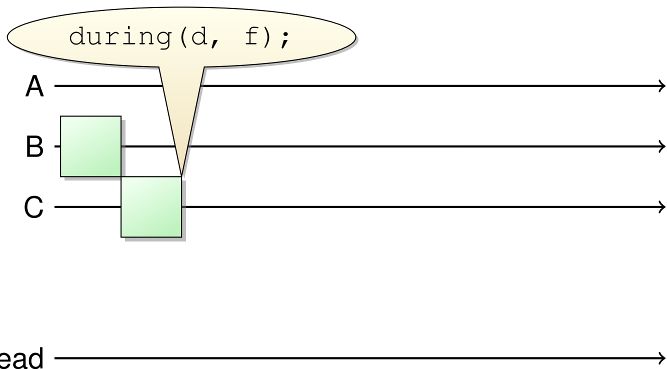
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
①   std::thread t(f); // Thread creation
②   sc_core::wait(d); // SystemC executes
③   t.join(); // Wait for completion
}
```



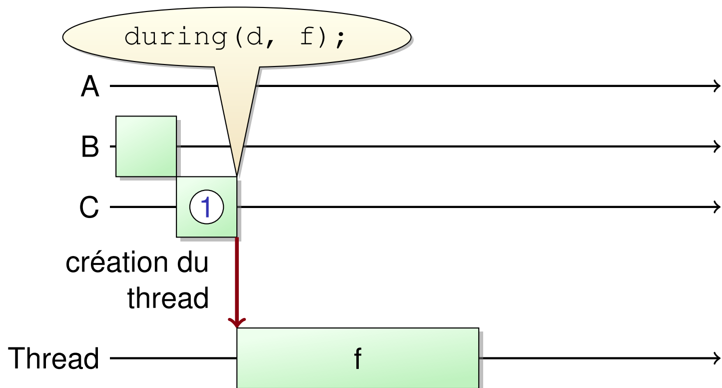
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
①    std::thread t(f); // Thread creation
②    sc_core::wait(d); // SystemC executes
③    t.join(); // Wait for completion
}
```



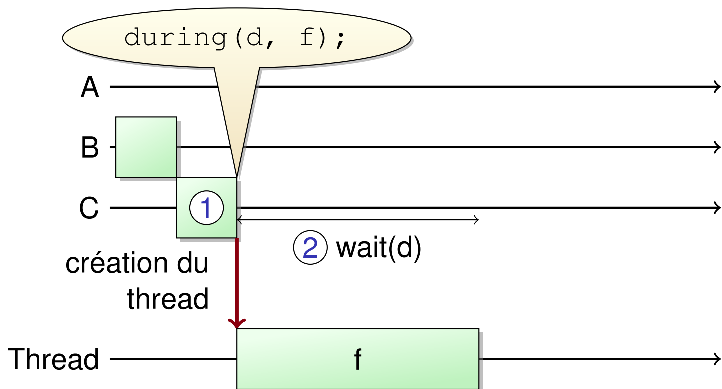
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
①   std::thread t(f); // Thread creation
②   sc_core::wait(d); // SystemC executes
③   t.join(); // Wait for completion
}
```



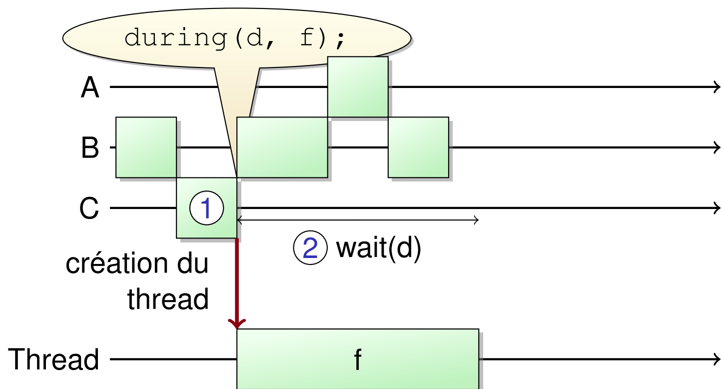
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
①    std::thread t(f); // Thread creation
②    sc_core::wait(d); // SystemC executes
③    t.join(); // Wait for completion
}
```



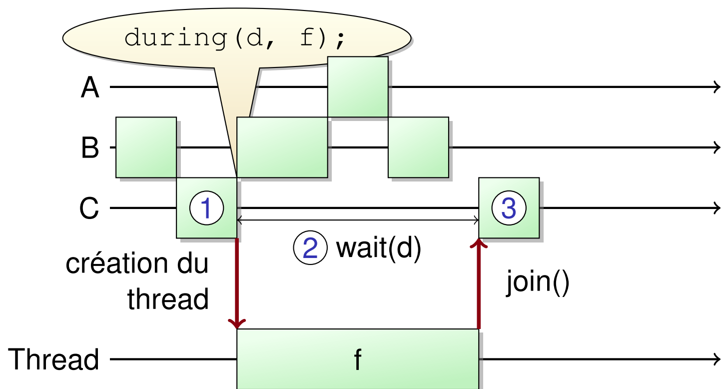
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
①    std::thread t(f); // Thread creation
②    sc_core::wait(d); // SystemC executes
③    t.join(); // Wait for completion
}
```



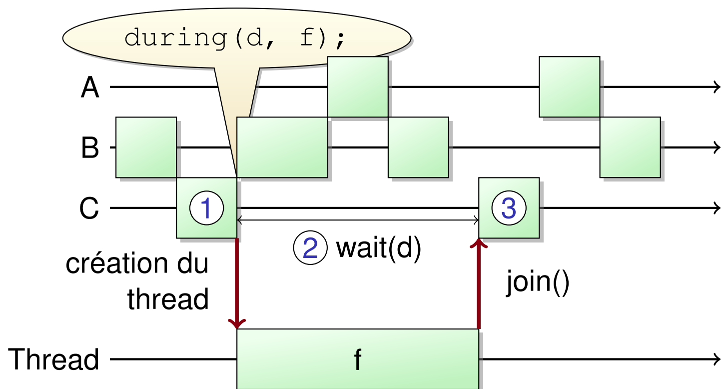
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
①   std::thread t(f); // Thread creation
②   sc_core::wait(d); // SystemC executes
③   t.join(); // Wait for completion
}
```



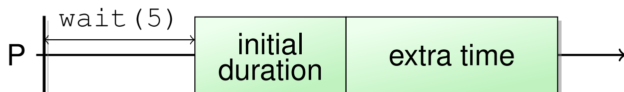
SC-DURING: First (Naive) Implementation

```
void during(sc_core::sc_time d,
            std::function<void()> f) {
    std::thread t(f); // Thread creation
    sc_core::wait(d); // SystemC executes
    t.join(); // Wait for completion
}
```



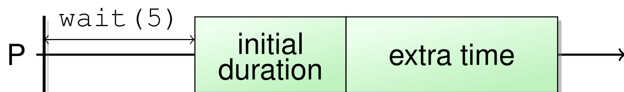
SC-DURING: New Synchronization Primitives

extra_time(t): Increase duration of current task



SC-DURING: New Synchronization Primitives

extra_time(t): Increase duration of current task

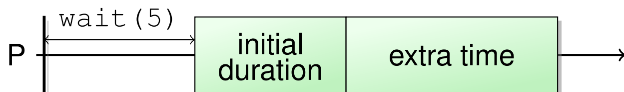


catch_up(): Wait for SystemC to reach the end of the task

```
while (!c) {
    extra_time(10);
    catch_up(); // Ensures fairness
}
```

SC-DURING: New Synchronization Primitives

extra_time(t): Increase duration of current task



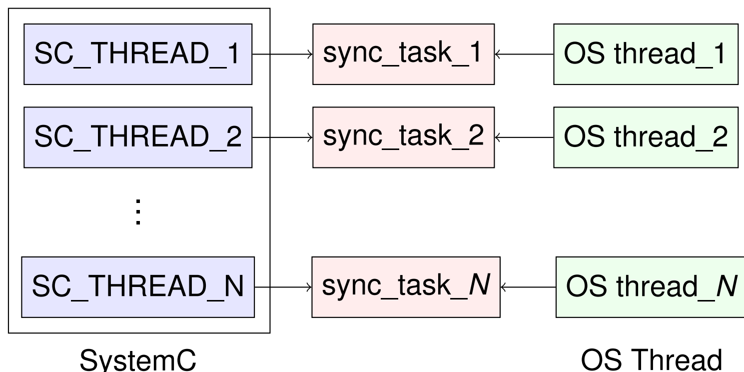
catch_up(): Wait for SystemC to reach the end of the task

```
while (!c) {
     extra_time(10);
     catch_up(); // Ensures fairness
}
```

sc_call(f): Call function `f` in the context of SystemC

```
x++; // Forbidden in
     // sc-during task
sc_call([]{  x++;  }); // OK
```

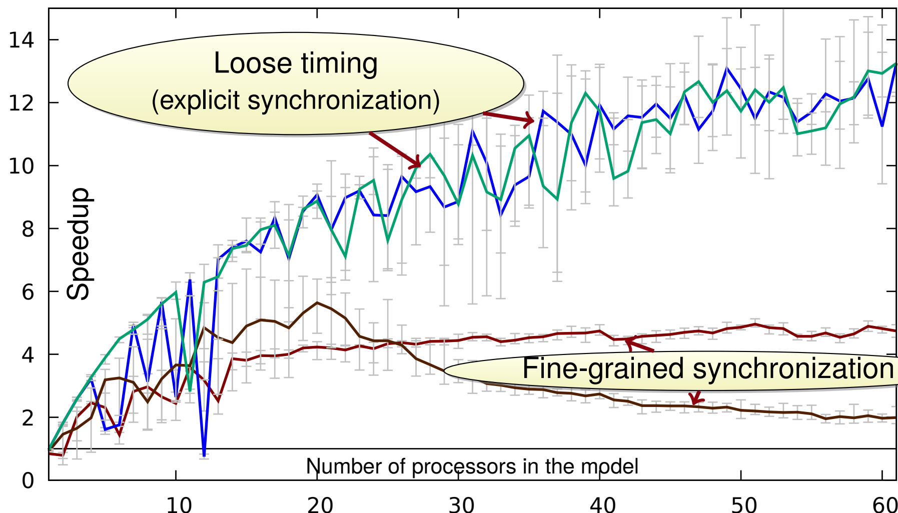
SC-DURING: Implementations



Strategies:

SEQ	Sequential (= reference)
THREAD	Thread creation + destruction for each task
POOL	Pre-allocated set of threads
ONDEMAND	Thread created on demand and reused

SC-DURING: Results



Test machine : $4 \times 12 = 48$ cores

Addressing the Faithfulness Issue: Exposing Bugs

Example bug: mis-placed synchronization:

```
imgReady = true;          while(!imgReady)
wait(5, SC_US);               wait(1, SC_US);
writeIMG();          ||   wait(10, SC_US);
wait(10, SC_US);          readIMG();
```

$\Rightarrow$ bug never seen in simulation

Addressing the Faithfulness Issue: Exposing Bugs

Example bug: mis-placed synchronization:

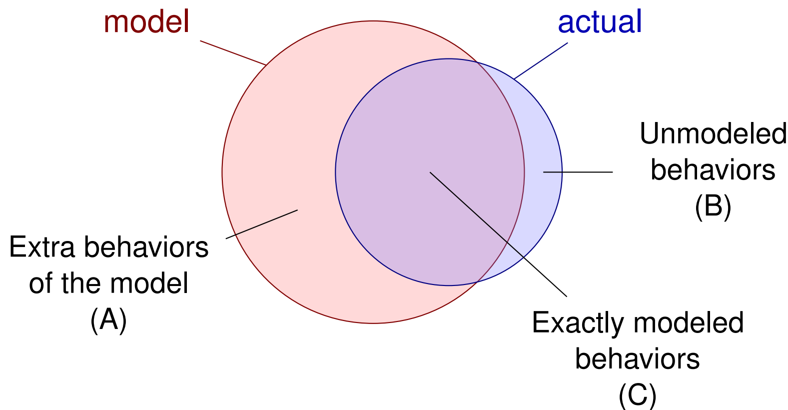
```
imgReady = true;          ||   while(!imgReady)
wait(5, SC_US);                    wait(1, SC_US);
writeIMG();                    wait(10, SC_US);
wait(10, SC_US);               readIMG();
```

⇒ bug never seen in simulation

```
during(15, SC_US, []{     ||   while(!imgReady)
    imgReady = true;               wait(1, SC_US);
    writeIMG();                wait(10, SC_US);
});                            readIMG();
```

⇒ strictly more behaviors, including the buggy one

Model Faithfulness



SC-DURING

- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed systems
- Exposes more bugs (⚠ faithfulness Vs correction)
- Next steps (skipped from this talk):
  - Worker threads Vs platform partitioning: DistemC
  - Exploit FIFO-based communication: FOFIFON
  - Integration in the design-flow: HLS code wrapping

This section

4. Extra-Functional Properties in TLM
   - Time and Parallelism
   - Power and Temperature Estimation

Power and Temperature Estimation

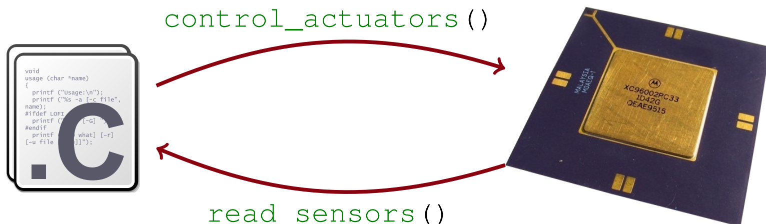
An example

"How to validate embedded software that regulates the chip's temperature?"

```
while (true) {
    // Temperature of one or more
    // locations of the chip
    read_sensors();

    compute();

    // Reduce frequency/voltage,
    // emergency stop, ...
    control_actuators();
}
```

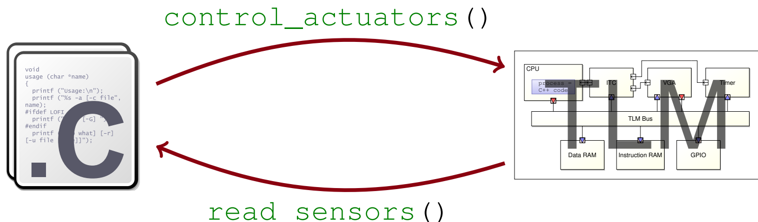
Power and Temperature Estimation

What precision? What applications?



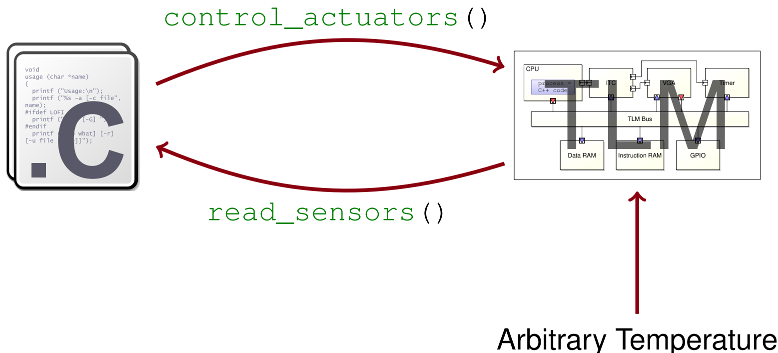
Power and Temperature Estimation

What precision? What applications?



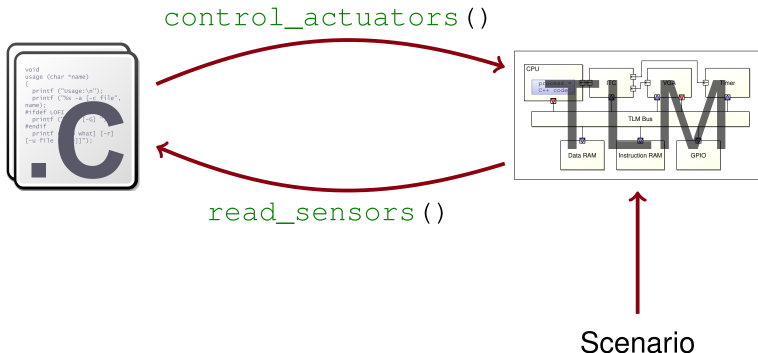
Power and Temperature Estimation

What precision? What applications?



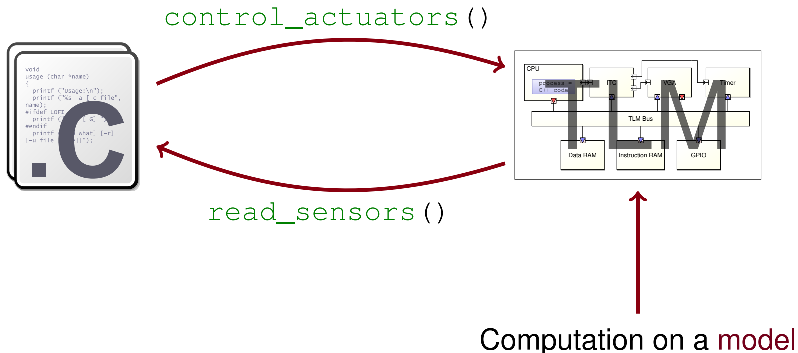
Power and Temperature Estimation

What precision? What applications?



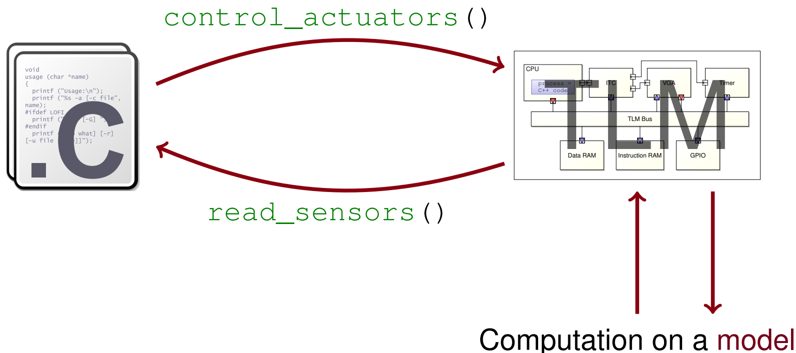
Power and Temperature Estimation

What precision? What applications?

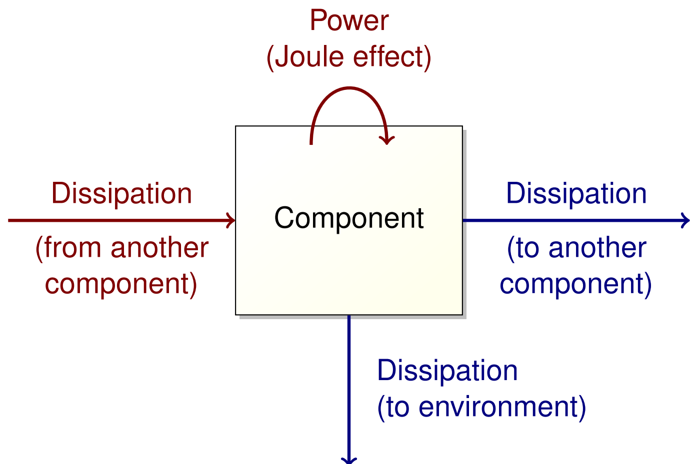


Power and Temperature Estimation

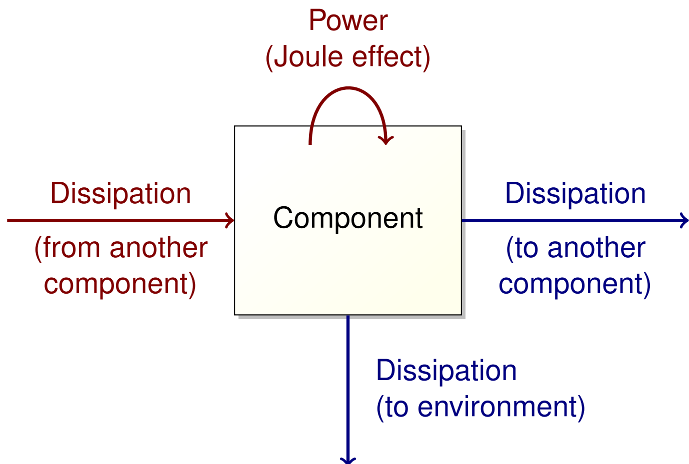
What precision? What applications?



Power Consumption, Temperature, Heat Dissipation

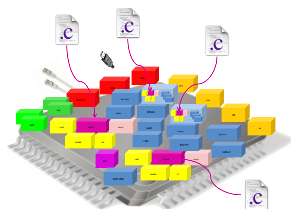


Power Consumption, Temperature, Heat Dissipation



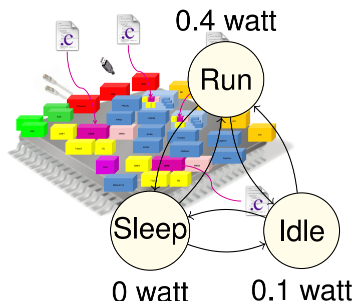
$\rightsquigarrow$ differential equations, solved by dedicated solvers

Estimation with Power-State Models



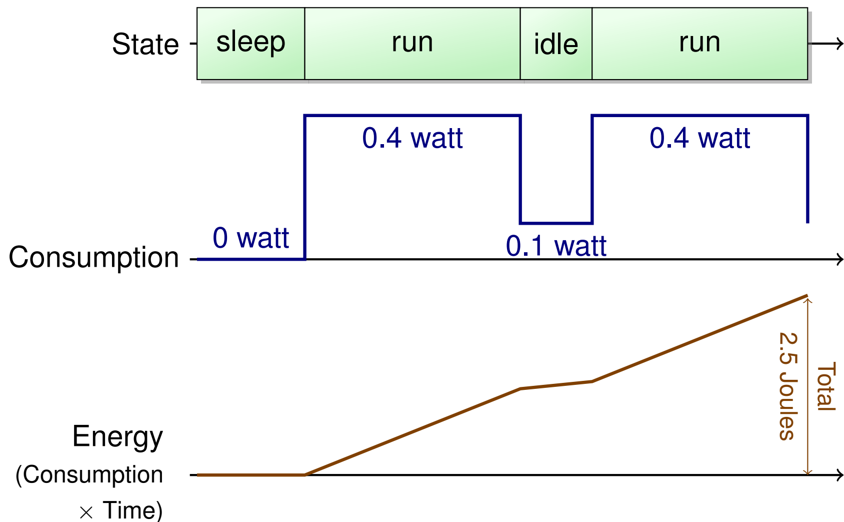
```
// SystemC Process
void compute() {
    while (true) {

        f();
        wait(10);

        wait();
    }
}
```

Estimation with Power-State Models

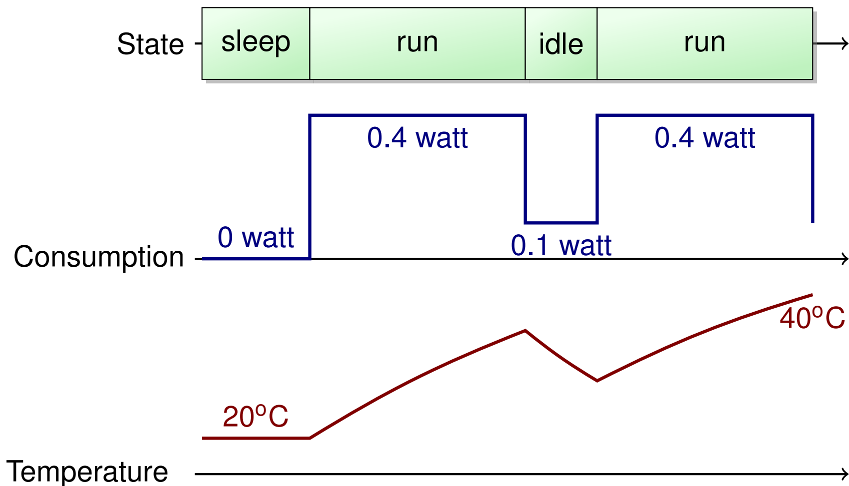



```
// SystemC Process
void compute() {
    while (true) {
        set_state("run");
        f();
        wait(10);
        set_state("idle");
        wait();
    }
}
```

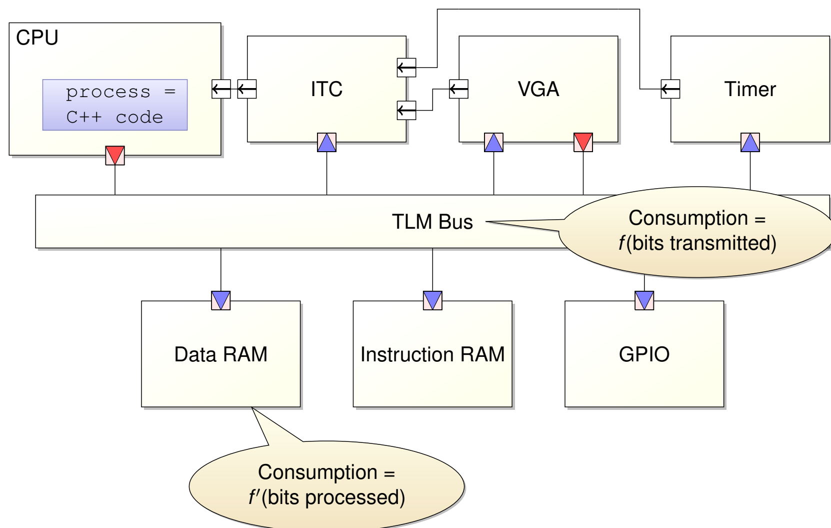
From States to Consumption



From Power to Temperature

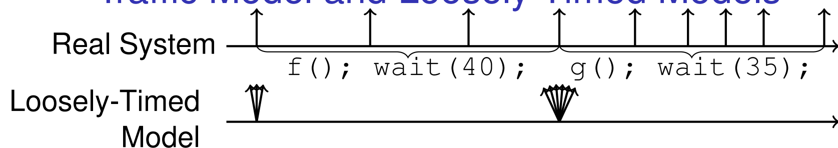


Traffic Models

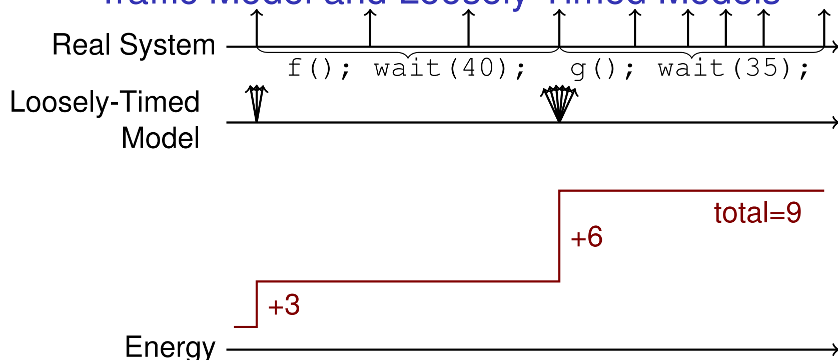


Traffic Model and Loosely Timed Models

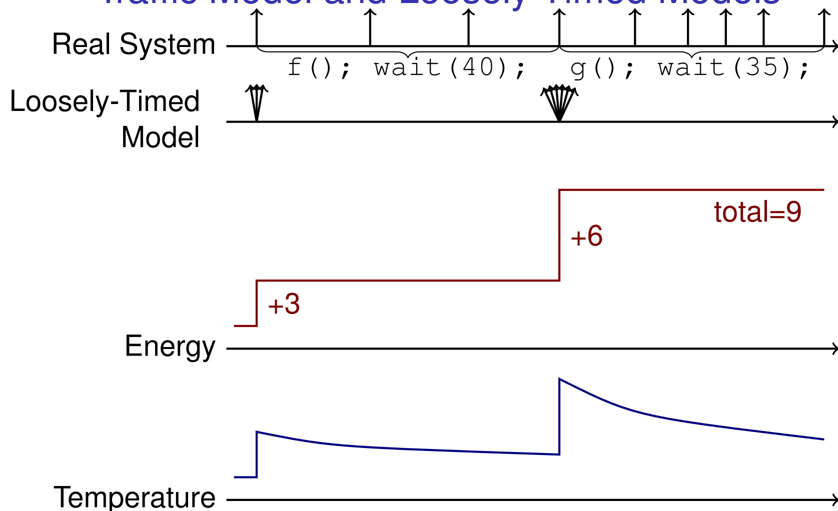
Traffic Model and Loosely Timed Models



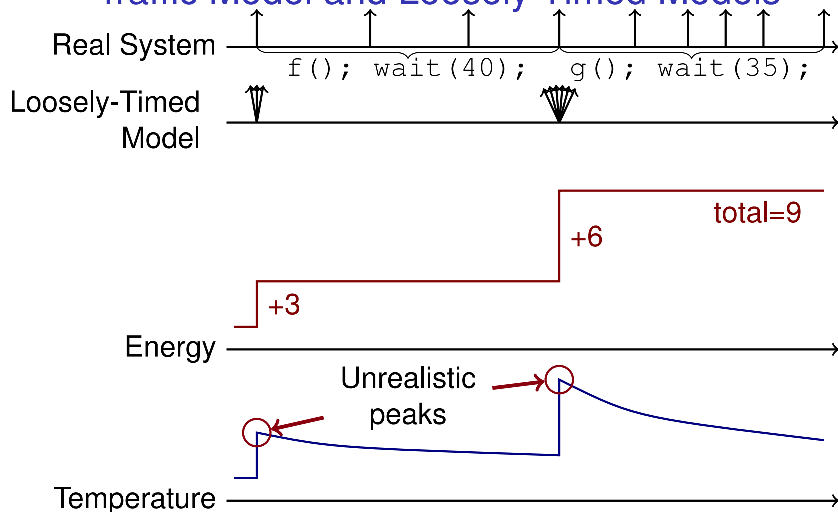
Traffic Model and Loosely Timed Models



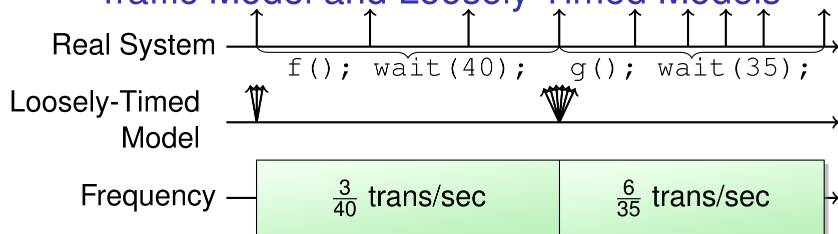
Traffic Model and Loosely Timed Models



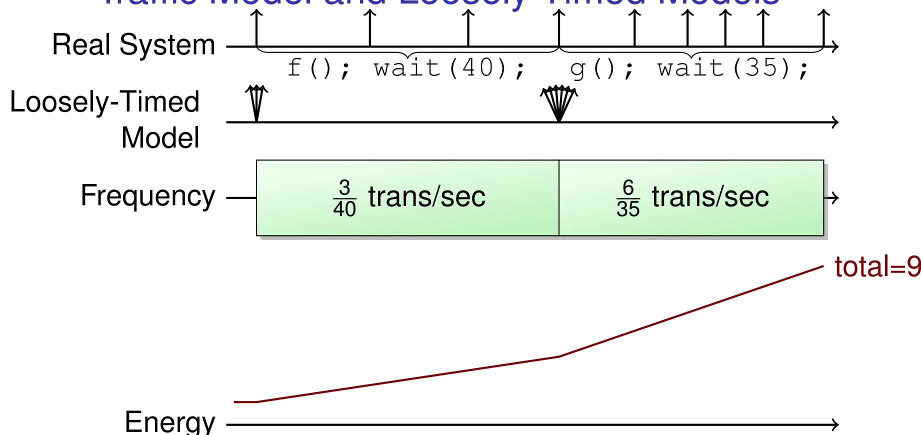
Traffic Model and Loosely Timed Models



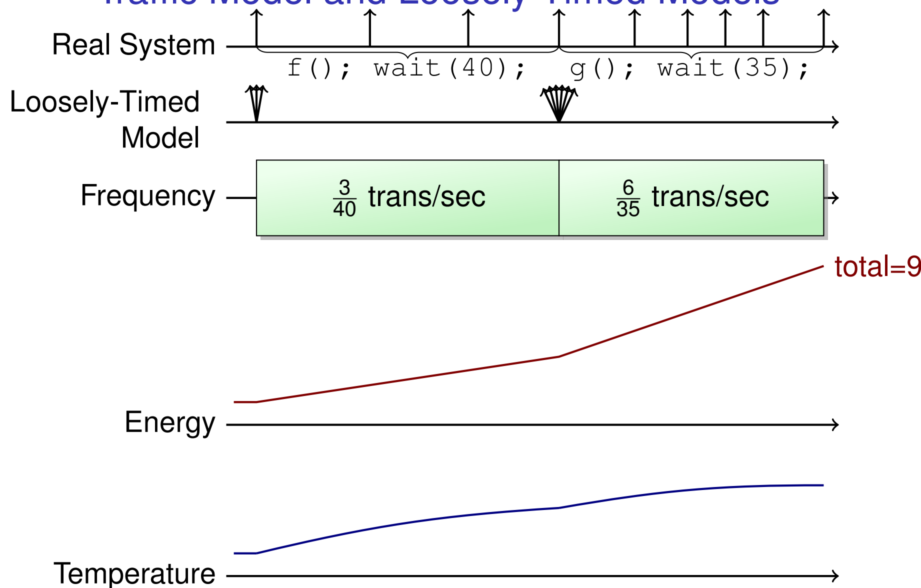
Traffic Model and Loosely Timed Models



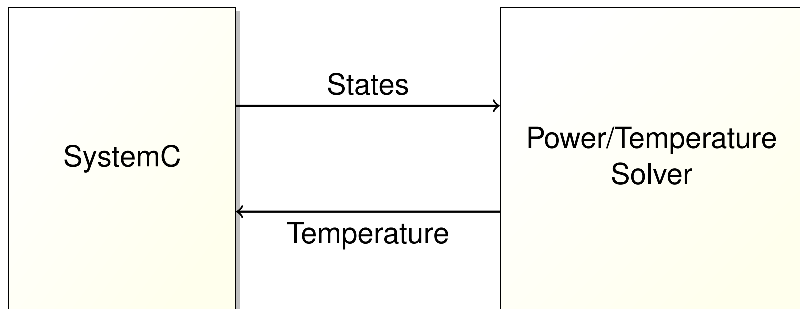
Traffic Model and Loosely Timed Models



Traffic Model and Loosely Timed Models

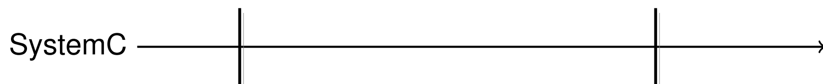


Cosimulation SystemC and Extra-Functional Solver

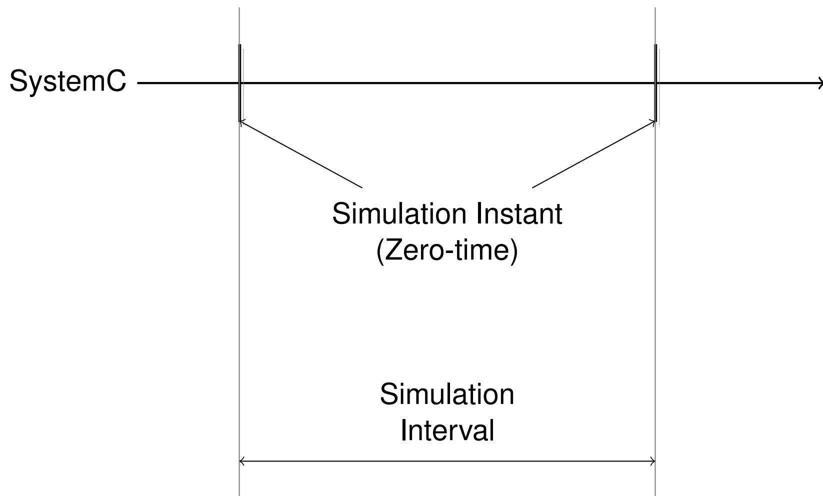


Functionality can depend on extra-functional data
(e.g.: temperature sensor)

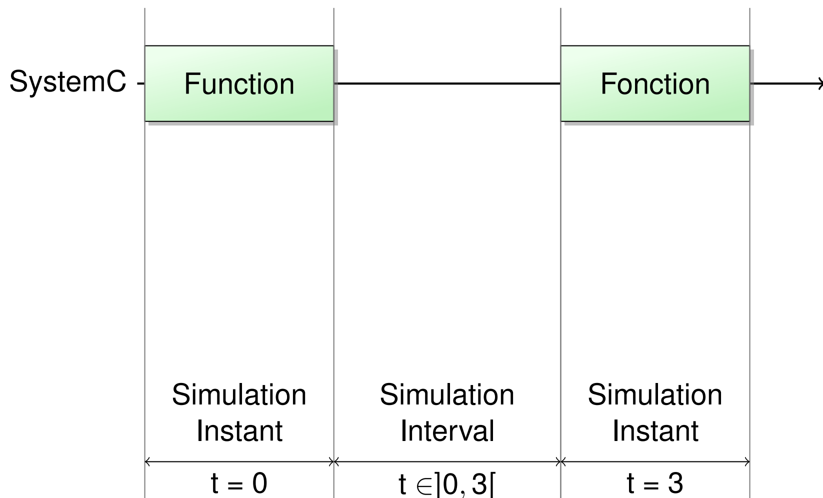
Cosimulation of SystemC and Extra-Functional Solver



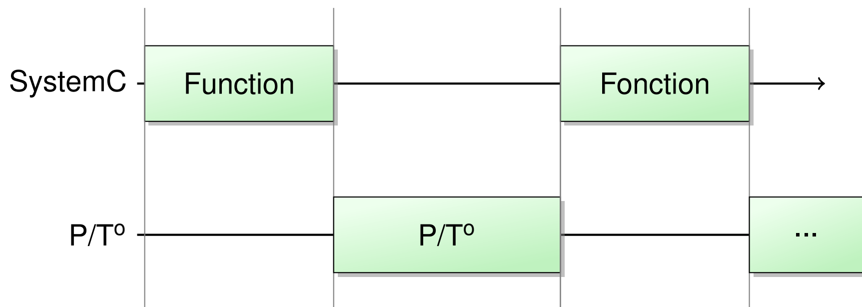
Cosimulation of SystemC and Extra-Functional Solver



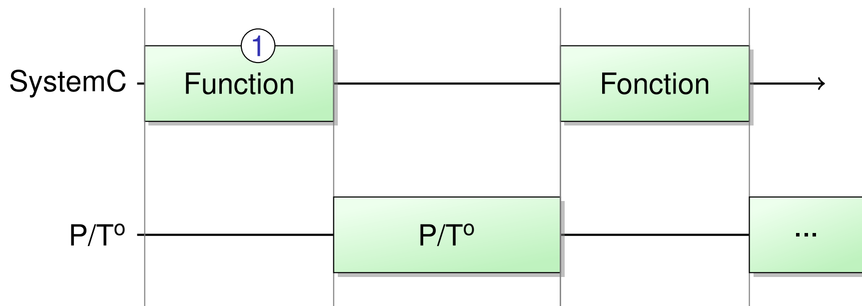
Cosimulation of SystemC and Extra-Functional Solver



Cosimulation of SystemC and Extra-Functional Solver

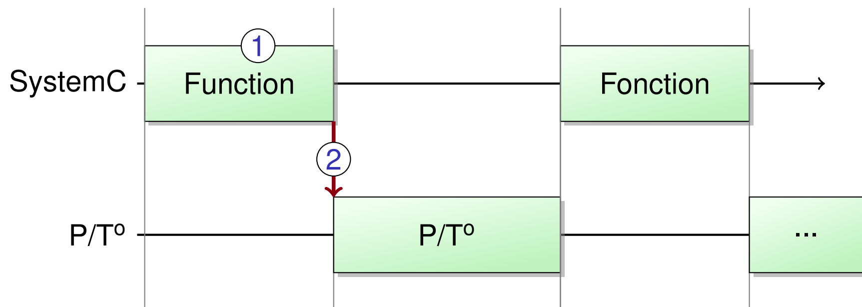


Cosimulation of SystemC and Extra-Functional Solver



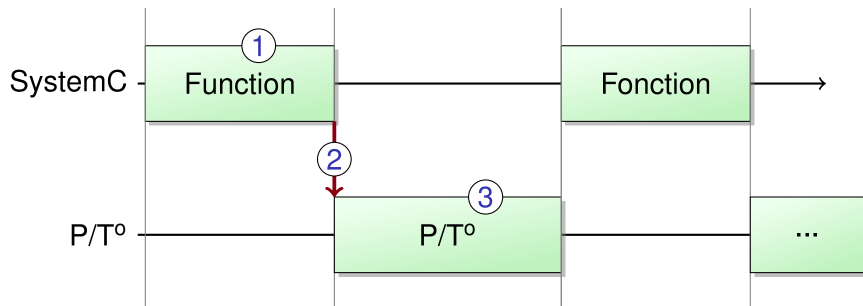
(1) SystemC runs simulation up to end of instant *t*

Cosimulation of SystemC and Extra-Functional Solver



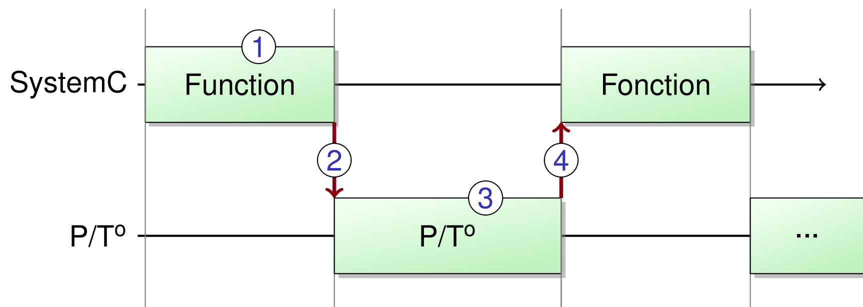
1. SystemC runs simulation up to end of instant t
2. SystemC sends a request for extra-functional simulation on $[t, t + d]$

Cosimulation of SystemC and Extra-Functional Solver



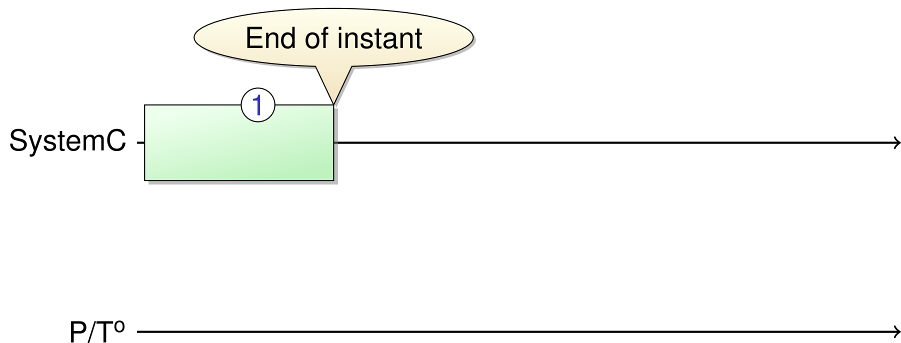
1. SystemC runs simulation up to end of instant t
2. SystemC sends a request for extra-functional simulation on $[t, t + d]$
3. Extra-functional solver does the computation on the interval

Cosimulation of SystemC and Extra-Functional Solver



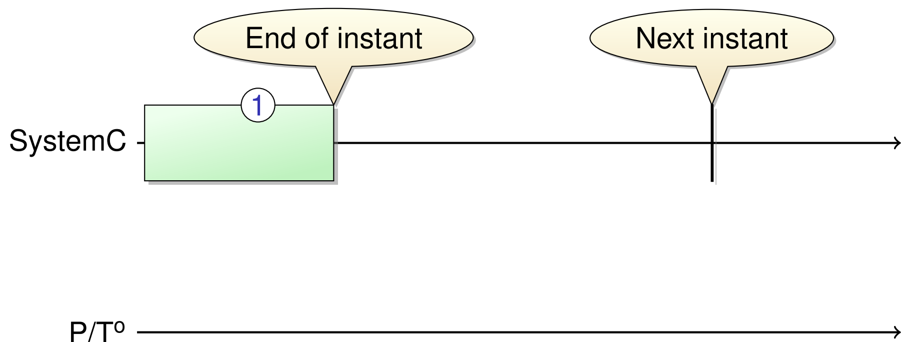
1. SystemC runs simulation up to end of instant t
2. SystemC sends a request for extra-functional simulation on $[t, t + d]$
3. Extra-functional solver does the computation on the interval
4. SystemC resumes simulation at beginning of instant $t + d$

Extra-Functional Events



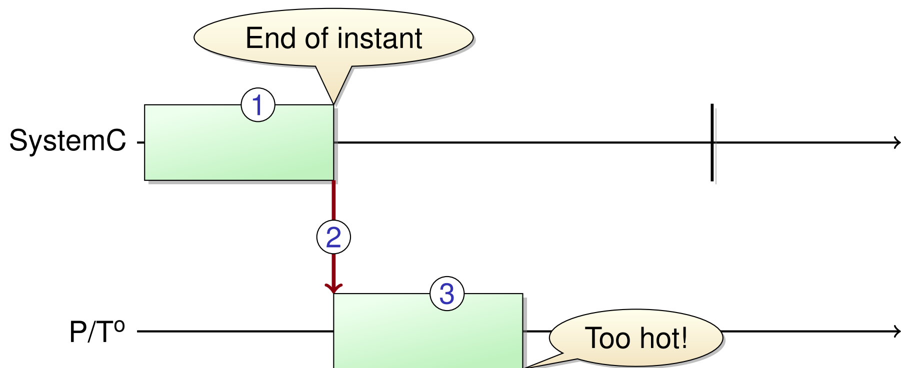
① SystemC runs simulation until end of instant *t*

Extra-Functional Events



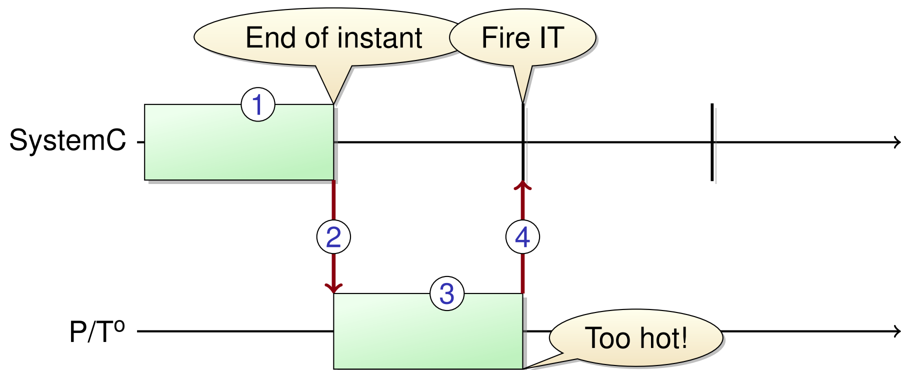
1. SystemC runs simulation until end of instant t
2. SystemC requests a extra-functional simulation in $[t, t+d]$ or until "too hot"

Extra-Functional Events



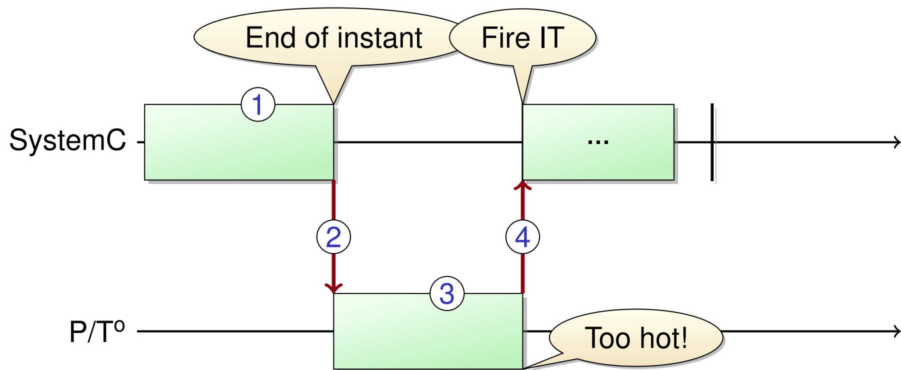
1. SystemC runs simulation until end of instant t
2. SystemC requests a extra-functional simulation in $[t, t + d]$ or until "too hot"
3. Extra-functional runs simulation, encounters stop condition

Extra-Functional Events



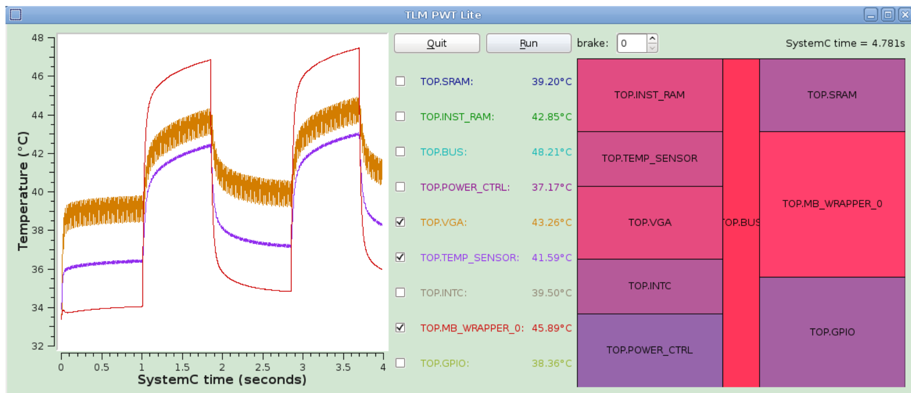
1. SystemC runs simulation until end of instant t
2. SystemC requests a extra-functional simulation in $[t, t + d]$ or until "too hot"
3. Extra-functional runs simulation, encounters stop condition
4. SystemC resumes earlier than expected with interrupt.

Extra-Functional Events



1. SystemC runs simulation until end of instant t
2. SystemC requests a extra-functional simulation in $[t, t + d]$ or until "too hot"
3. Extra-functional runs simulation, encounters stop condition
4. SystemC resumes earlier than expected with interrupt.

Results



Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion

Conclusion

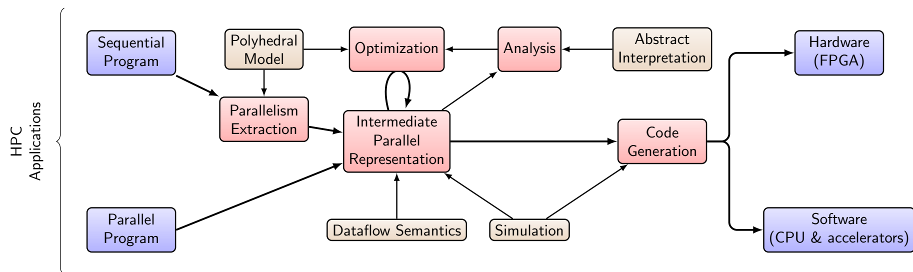
Transaction-Level Models of
Systems-on-a-Chip
Can they be
Fast, Correct and Faithful?

Conclusion

- Fast
  - Optimized compiler
  - Parallelization techniques
  - High abstraction level (Loose Timing)
- Correct
  - Formal verification
- Faithful
  - More ways to express concurrency
  - Preserve Faithfulness of Temperature Models for Loose Timing

The new CASH Team, LIP (ENS-Lyon)

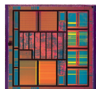
Compilation and
Analysis for
Software and
Hardware



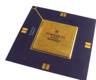
Christophe Alias, Laure Gonnord, Matthieu Moy

Questions?

Sources



`http://en.wikipedia.org/wiki/File:Diopsis.jpg`
(Peter John Bishop, CC Attribution-Share Alike 3.0 Unported)



`http://www.fotopedia.com/items/flickr-367843750`
(oskay@fotopedia, CC Attribution 2.0 Generic)